eBook Gratuit

APPRENEZ Python Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#python

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: python-language

It is an unofficial and free Python Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Python Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec le langage Python

Remarques

Python est un langage de programmation largement utilisé. C'est:

- **Haut niveau** : Python automatise les opérations de bas niveau telles que la gestion de la mémoire. Il laisse au programmeur un peu moins de contrôle mais présente de nombreux avantages, notamment la lisibilité du code et des expressions de code minimales.
- **Usage général** : Python est conçu pour être utilisé dans tous les contextes et environnements. Un exemple de langage non généraliste est PHP: il est conçu spécifiquement comme un langage de script de développement Web côté serveur. En revanche, Python *peut* être utilisé pour le développement Web côté serveur, mais également pour créer des applications de bureau.
- **Typé dynamiquement** : chaque variable de Python peut référencer n'importe quel type de données. Une seule expression peut évaluer des données de différents types à des moments différents. De ce fait, le code suivant est possible:

```
if something:
    x = 1
else:
    x = 'this is a string'
print(x)
```

- **Fortement typé** : Lors de l'exécution du programme, vous n'êtes pas autorisé à faire quoi que ce soit incompatible avec le type de données avec lequel vous travaillez. Par exemple, il n'y a pas de conversions cachées de chaînes en nombres; une chaîne composée de chiffres ne sera jamais traitée comme un nombre sauf si vous la convertissez explicitement:

```
1 + '1'  # raises an error
1 + int('1')  # results with 2
```

- **Amical pour les débutants :)** : La syntaxe et la structure de Python sont très intuitives. Il est de haut niveau et fournit des concepts destinés à permettre l'écriture de programmes clairs à la fois à petite et grande échelle. Python prend en charge plusieurs paradigmes de programmation, y compris la programmation orientée objet, impérative et fonctionnelle ou les styles procéduraux. Il dispose d'une bibliothèque standard complète et de nombreuses bibliothèques tierces faciles à installer.

Ses principes de conception sont décrits dans *The Zen of Python* .

Actuellement, il existe deux branches principales de Python qui présentent des différences

significatives. Python 2.x est la version héritée même si son utilisation est encore répandue. Python 3.x effectue un ensemble de modifications incompatibles avec les versions antérieures qui visent à réduire la duplication des fonctionnalités. Pour vous aider à choisir la version qui vous convient le mieux, consultez cet article .

La documentation officielle de Python est également une ressource complète et utile, contenant de la documentation pour toutes les versions de Python, ainsi que des didacticiels pour vous aider à démarrer.

Il existe une implémentation officielle du langage fournie par Python.org, généralement appelée CPython, et plusieurs implémentations alternatives du langage sur d'autres plates-formes d'exécution. Il s'agit notamment d' IronPython (exécutant Python sur la plate-forme .NET), Jython (sur le runtime Java) et PyPy (implémentant Python dans un sous-ensemble de lui-même).

Versions

Python 3.x

Version	Date de sortie
[3.7]	2017-05-08
3.6	2016-12-23
3.5	2015-09-13
3.4	2014-03-17
3.3	2012-09-29
3.2	2011-02-20
3.1	2009-06-26
3.0	2008-12-03

Python 2.x

Version	Date de sortie
2.7	2010-07-03
2.6	2008-10-02
2,5	2006-09-19
2.4	2004-11-30

Version	Date de sortie
2.3	2003-07-29
2.2	2001-12-21
2.1	2001-04-15
2.0	2000-10-16

Examples

Commencer

Python est un langage de programmation de haut niveau largement utilisé pour la programmation à usage général, créé par Guido van Rossum et publié pour la première fois en 1991. Python intègre un système de type dynamique et une gestion automatique de la mémoire. programmation fonctionnelle et styles procéduraux. Il a une bibliothèque standard large et complète.

Deux versions principales de Python sont actuellement utilisées:

- Python 3.x est la version actuelle et est en cours de développement.
- Python 2.x est la version héritée et ne recevra que les mises à jour de sécurité jusqu'en 2020. Aucune nouvelle fonctionnalité ne sera implémentée. Notez que de nombreux projets utilisent encore Python 2, bien que la migration vers Python 3 devienne plus facile.

Vous pouvez télécharger et installer les deux versions de Python ici . Voir Python 3 vs Python 2 pour une comparaison entre eux. De plus, certaines tierces parties proposent des versions reconditionnées de Python qui ajoutent des bibliothèques couramment utilisées et d'autres fonctionnalités pour faciliter la configuration des cas d'utilisation courants, tels que les mathématiques, l'analyse de données ou l'utilisation scientifique. Voir la liste sur le site officiel .

Vérifiez si Python est installé

Pour confirmer que Python a été installé correctement, vous pouvez le vérifier en exécutant la commande suivante dans votre terminal favori (si vous utilisez le système d'exploitation Windows, vous devez ajouter le chemin de python à la variable d'environnement avant de l'utiliser dans l'invite de commande):

```
$ python --version
```

Python 3.x 3.0

Si vous avez installé *Python 3* et qu'il s'agit de votre version par défaut (voir **Dépannage** pour plus de détails), vous devriez voir quelque chose comme ceci:

```
$ python --version
Python 3.6.0
```

Python 2.x 2.7

Si vous avez installé *Python 2* et qu'il s'agit de votre version par défaut (voir **Dépannage** pour plus de détails), vous devriez voir quelque chose comme ceci:

```
$ python --version
Python 2.7.13
```

Si vous avez installé Python 3, mais que `$ python --version` affiche une version Python 2, Python 2 est également installé. C'est souvent le cas sur MacOS et de nombreuses distributions Linux. Utilisez plutôt `$ python3` pour utiliser explicitement l'interpréteur Python 3.

Bonjour, World in Python en utilisant IDLE

IDLE est un éditeur simple pour Python, fourni avec Python.

Comment créer Hello, programme mondial dans IDLE

- Ouvrez IDLE sur votre système de choix.
  - Dans les anciennes versions de Windows, il se trouve dans `All Programs` du menu Windows.
  - Dans Windows 8+, recherchez `IDLE` ou recherchez-le dans les applications présentes sur votre système.
  - Sur les systèmes Unix (y compris Mac), vous pouvez l'ouvrir à partir du shell en tapant `$ idle python_file.py` .
- Il ouvrira un shell avec des options en haut.

Dans le shell, il y a un prompt de trois parenthèses à angle droit:

```
>>>
```

Maintenant, écrivez le code suivant dans l'invite:

```
>>> print("Hello, World")
```

Appuyez sur `Entrée` .

```
>>> print("Hello, World")
Hello, World
```

Fichier Python Hello World

Créez un nouveau fichier `hello.py` contenant la ligne suivante:

Python 3.x 3.0

```
print('Hello, World')
```

Python 2.x 2.6

Vous pouvez utiliser la fonction d' `print` Python 3 dans Python 2 avec l'instruction d' `import` suivante:

```
from __future__ import print_function
```

Python 2 possède un certain nombre de fonctionnalités qui peuvent être importées à partir de Python 3 à l'aide du module `__future__` , comme indiqué ici .

Python 2.x 2.7

Si vous utilisez Python 2, vous pouvez également taper la ligne ci-dessous. Notez que ceci n'est pas valide dans Python 3 et n'est donc pas recommandé car il réduit la compatibilité du code inter-versions.

```
print 'Hello, World'
```

Dans votre terminal, accédez au répertoire contenant le fichier `hello.py` .

Tapez `python hello.py` , puis appuyez sur la touche `Entrée` .

```
$ python hello.py
Hello, World
```

Vous devriez voir `Hello, World` imprimé sur la console.

Vous pouvez également remplacer `hello.py` par le chemin d'accès à votre fichier. Par exemple, si vous avez le fichier dans votre répertoire personnel et que votre utilisateur est "utilisateur" sous Linux, vous pouvez taper `python /home/user/hello.py` .

Lancer un shell Python interactif

En exécutant (exécutant) la commande `python` dans votre terminal, vous obtenez un shell Python interactif. Ceci est également connu sous le nom d' Interpréteur Python ou REPL (pour 'Read Evaluate Print Loop').

```
$ python
Python 2.7.12 (default, Jun 28 2016, 08:46:01)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello, World'
Hello, World
>>>
```

Si vous souhaitez exécuter Python 3 à partir de votre terminal, exécutez la commande `python3` .

```
$ python3
Python 3.6.0 (default, Jan 13 2017, 00:00:00)
[GCC 6.1.1 20160602] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello, World')
Hello, World
>>>
```

Vous pouvez également lancer l'invite interactive et charger le fichier avec `python -i <file.py>` .

En ligne de commande, exécutez:

```
$ python -i hello.py
"Hello World"
>>>
```

Il existe plusieurs façons de fermer le shell Python:

```
>>> exit()
```

ou

```
>>> quit()
```

Alternativement, `CTRL + D` va fermer le shell et vous remettre sur la ligne de commande de votre terminal.

Si vous souhaitez annuler une commande en cours de saisie et revenir à une invite de commande propre, tout en restant dans le shell Interpreter, utilisez `CTRL + C`.

Essayez un shell Python interactif en ligne .

Autres coquilles en ligne

Divers sites Web offrent un accès en ligne aux shells Python.

Les shells en ligne peuvent être utiles aux fins suivantes:

- Exécutez un petit extrait de code à partir d'une machine dépourvue d'installation python (smartphones, tablettes, etc.).
- Apprendre ou enseigner le Python de base.
- Résoudre des problèmes de juge en ligne.

Exemples:

> Déni de responsabilité: les auteurs de la documentation ne sont affiliés à aucune des ressources énumérées ci-dessous.

- https://www.python.org/shell/ - Le shell en ligne Python hébergé par le site Web officiel de

Python.

- https://ideone.com/ - Largement utilisé sur le Net pour illustrer le comportement des extraits de code.
- https://repl.it/languages/python3 - Compilateur en ligne puissant et simple, IDE et interpréteur. Code, compiler et exécuter du code en Python.
- https://www.tutorialspoint.com/execute_python_online.php - Shell UNIX complet et explorateur de projet convivial.
- http://rextester.com/l/python3_online_compiler - IDE simple et facile à utiliser qui affiche le temps d'exécution

Exécuter des commandes sous forme de chaîne

Python peut être passé du code arbitraire sous forme de chaîne dans le shell:

```
$ python -c 'print("Hello, World")'
Hello, World
```

Cela peut être utile lors de la concaténation des résultats de scripts dans le shell.

Coquillages et au-delà

Gestion des packages - L'outil recommandé par PyPA pour installer les packages Python est PIP . Pour l'installer, sur la ligne de commande, exécutez `pip install <the package name>` . Par exemple, `pip install numpy` . (Remarque: sur Windows, vous devez ajouter pip à vos variables d'environnement PATH. Pour éviter cela, utilisez `python -m pip install <the package name>`)

Shells - Jusqu'à présent, nous avons discuté de différentes manières d'exécuter du code en utilisant le shell interactif natif de Python. Les coquilles utilisent le pouvoir d'interprétation de Python pour expérimenter le code en temps réel. Les shells alternatifs incluent IDLE - une interface graphique pré-intégrée, IPython - connu pour étendre l'expérience interactive, etc.

Programmes - Pour le stockage à long terme, vous pouvez enregistrer le contenu dans des fichiers .py et les éditer / exécuter en tant que scripts ou programmes avec des outils externes tels que shell, IDE (PyCharm), ordinateurs portables Jupyter , etc. Cependant, les méthodes présentées ici sont suffisantes pour commencer.

Python tutor vous permet de parcourir le code Python afin de visualiser le déroulement du programme et de vous aider à comprendre où votre programme a mal tourné.

PEP8 définit des directives pour le formatage du code Python. Le formatage du code est important pour que vous puissiez lire rapidement le code.

Création de variables et affectation de valeurs

Pour créer une variable en Python, il vous suffit de spécifier le nom de la variable, puis de lui attribuer une valeur.

```
<variable name> = <value>
```

Python utilise = pour attribuer des valeurs aux variables. Il n'est pas nécessaire de déclarer une variable à l'avance (ou de lui attribuer un type de données), l'affectation d'une valeur à une variable elle-même déclare et initialise la variable avec cette valeur. Il n'y a aucun moyen de déclarer une variable sans lui attribuer une valeur initiale.

```
# Integer
a = 2
print(a)
# Output: 2

# Integer
b = 9223372036854775807
print(b)
# Output: 9223372036854775807

# Floating point
pi = 3.14
print(pi)
# Output: 3.14

# String
c = 'A'
print(c)
# Output: A

# String
name = 'John Doe'
print(name)
# Output: John Doe

# Boolean
q = True
print(q)
# Output: True

# Empty value or null data type
x = None
print(x)
# Output: None
```

L'attribution des variables fonctionne de gauche à droite. Donc, ce qui suit vous donnera une erreur de syntaxe.

```
0 = x
=> Output: SyntaxError: can't assign to literal
```

Vous ne pouvez pas utiliser les mots-clés de python comme nom de variable valide. Vous pouvez voir la liste des mots clés par:

```
import keyword
```

```
print(keyword.kwlist)
```

Règles de nommage des variables:

1. Les noms de variables doivent commencer par une lettre ou un trait de soulignement.

```
 x  = True   # valid
 _y = True   # valid

 9x = False  # starts with numeral
 => SyntaxError: invalid syntax

 $y = False #  starts with symbol
 => SyntaxError: invalid syntax
```

2. Le reste de votre nom de variable peut être composé de lettres, de chiffres et de traits de soulignement.

```
has_0_in_it = "Still Valid"
```

3. Les noms sont sensibles à la casse.

```
x = 9
y = X*5
=>NameError: name 'X' is not defined
```

Même s'il n'est pas nécessaire de spécifier un type de données lors de la déclaration d'une variable en Python, tout en allouant la zone nécessaire en mémoire à la variable, l'interpréteur Python sélectionne automatiquement le type intégré le plus approprié:

```
a = 2
print(type(a))
# Output: <type 'int'>

b = 9223372036854775807
print(type(b))
# Output: <type 'int'>

pi = 3.14
print(type(pi))
# Output: <type 'float'>

c = 'A'
print(type(c))
# Output: <type 'str'>

name = 'John Doe'
print(type(name))
# Output: <type 'str'>

q = True
print(type(q))
# Output: <type 'bool'>
```

```
x = None
print(type(x))
# Output: <type 'NoneType'>
```

Maintenant que vous connaissez les bases de l'assignation, voyons cette subtilité à propos de l'affectation en python.

Lorsque vous utilisez `=` pour effectuer une opération d'affectation, ce qui est à gauche de `=` est un ***nom*** pour l' ***objet*** à droite. Enfin, quel `=` fait est assigner la ***référence*** de l'objet sur le droit au ***nom*** de la gauche.

C'est:

```
a_name = an_object  # "a_name" is now a name for the reference to the object "an_object"
```

Donc, à partir de nombreux exemples d'affectation ci-dessus, si nous choisissons `pi = 3.14` , alors `pi` est **un** nom (pas **le** nom, car un objet peut avoir plusieurs noms) pour l'objet `3.14` . Si vous ne comprenez pas quelque chose ci-dessous, revenez à ce point et lisez-le à nouveau! En outre, vous pouvez jeter un oeil à ce pour une meilleure compréhension.

Vous pouvez affecter plusieurs valeurs à plusieurs variables sur une seule ligne. Notez qu'il doit y avoir le même nombre d'arguments sur les côtés droit et gauche de l'opérateur `=` :

```
a, b, c = 1, 2, 3
print(a, b, c)
# Output: 1 2 3

a, b, c = 1, 2
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b, c = 1, 2
=> ValueError: need more than 2 values to unpack

a, b = 1, 2, 3
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>     a, b = 1, 2, 3
=> ValueError: too many values to unpack
```

L'erreur dans le dernier exemple peut être évitée en attribuant des valeurs restantes à un nombre égal de variables arbitraires. Cette variable factice peut avoir n'importe quel nom, mais il est habituel d'utiliser le trait de soulignement (_) pour attribuer des valeurs indésirables:

```
a, b, _ = 1, 2, 3
print(a, b)
# Output: 1, 2
```

Notez que le nombre de _ et le nombre de valeurs restantes doivent être égaux. Sinon, "trop de valeurs pour décompresser l'erreur" sont renvoyées comme ci-dessus:

```
a, b, _ = 1,2,3,4
=>Traceback (most recent call last):
=>File "name.py", line N, in <module>
=>a, b, _ = 1,2,3,4
=>ValueError: too many values to unpack (expected 3)
```

Vous pouvez également affecter une valeur unique à plusieurs variables simultanément.

```
a = b = c = 1
print(a, b, c)
# Output: 1 1 1
```

Lors de l'utilisation d'une telle assignation en cascade, il est important de noter que les trois variables `a` , `b` et `c` font référence au *même objet* en mémoire, un objet `int` ayant la valeur 1. En d'autres termes, `a` , `b` et `c` sont trois noms différents donné au même objet int. L'attribution d'un objet différent à l'un d'eux ne modifie pas les autres, comme prévu:

```
a = b = c = 1    # all three names a, b and c refer to same int object with value 1
print(a, b, c)
# Output: 1 1 1
b = 2            # b now refers to another int object, one with a value of 2
print(a, b, c)
# Output: 1 2 1  # so output is as expected.
```

Ce qui précède est également vrai pour les types mutables (tels que `list` , `dict` , etc.) comme pour les types immuables (comme `int` , `string` , `tuple` , etc.):

```
x = y = [7, 8, 9]   # x and y refer to the same list object just created, [7, 8, 9]
x = [13, 8, 9]      # x now refers to a different list object just created, [13, 8, 9]
print(y)            # y still refers to the list it was first assigned
# Output: [7, 8, 9]
```

Jusqu'ici tout va bien. Les choses sont un peu différentes en ce qui concerne la *modification de* l'objet (contrairement à l' *attribution* du nom à un objet différent, ce que nous avons fait précédemment) lorsque l'affectation en cascade est utilisée pour les types mutables. Jetez un oeil ci-dessous, et vous le verrez de première main:

```
x = y = [7, 8, 9]     # x and y are two different names for the same list object just created,
[7, 8, 9]
x[0] = 13             # we are updating the value of the list [7, 8, 9] through one of its
names, x in this case
print(y)              # printing the value of the list using its other name
# Output: [13, 8, 9]  # hence, naturally the change is reflected
```

Les listes imbriquées sont également valables en python. Cela signifie qu'une liste peut contenir une autre liste en tant qu'élément.

```
x = [1, 2, [3, 4, 5], 6, 7] # this is nested list
print x[2]
# Output: [3, 4, 5]
print x[2][1]
```

```
# Output: 4
```

Enfin, les variables en Python ne doivent pas nécessairement rester du même type que celles définies pour la première fois - vous pouvez simplement utiliser `=` pour attribuer une nouvelle valeur à une variable, même si cette valeur est d'un type différent.

```
a = 2
print(a)
# Output: 2

a = "New value"
print(a)
# Output: New value
```

Si cela vous dérange, pensez au fait que ce qui est à gauche de `=` est juste un nom pour un objet. D'abord, vous appelez l'objet `int` avec la valeur 2 `a` , puis vous changez d'avis et décidez de donner le nom `a` à un objet `string` ayant la valeur 'Nouvelle valeur'. Simple, non?

Entrée utilisateur

Saisie interactive

Pour obtenir la contribution de l'utilisateur, utilisez la fonction `input` (**remarque** : dans Python 2.x, la fonction s'appelle `raw_input` place, bien que Python 2.x ait sa propre version d' `input` complètement différente):

Python 2.x 2.3

```
name = raw_input("What is your name? ")
# Out: What is your name? _
```

> **Remarque sur la sécurité** N'utilisez pas `input()` dans Python2 - le texte saisi sera évalué comme s'il s'agissait d'une expression Python (équivalent à `eval(input())` dans Python3), ce qui pourrait facilement devenir une vulnérabilité. Voir cet article pour plus d'informations sur les risques liés à l'utilisation de cette fonction.

Python 3.x 3.0

```
name = input("What is your name? ")
# Out: What is your name? _
```

Le reste de cet exemple utilisera la syntaxe Python 3.

La fonction prend un argument de chaîne, qui l'affiche sous la forme d'une invite et renvoie une chaîne. Le code ci-dessus fournit une invite, attendant que l'utilisateur saisisse.

```
name = input("What is your name? ")
# Out: What is your name?
```

Si l'utilisateur tape "Bob" et frappe, entrez, le `name` la variable sera assigné à la chaîne `"Bob"` :

```
name = input("What is your name? ")
# Out: What is your name? Bob
print(name)
# Out: Bob
```

Notez que l' `input` est toujours de type `str` , ce qui est important si vous souhaitez que l'utilisateur entre des nombres. Par conséquent, vous devez convertir le `str` avant d'essayer de l'utiliser comme un nombre:

```
x = input("Write a number:")
# Out: Write a number: 10
x / 2
# Out: TypeError: unsupported operand type(s) for /: 'str' and 'int'
float(x) / 2
# Out: 5.0
```

NB: Il est recommandé d'utiliser des blocs `try` / `except` pour intercepter des exceptions lors de l'utilisation d'entrées utilisateur . Par exemple, si votre code souhaite `raw_input` un `raw_input` en un `int` et que ce que l'écrit est impossible à graver, il déclenche une `ValueError` .

IDLE - Python GUI

IDLE est l'environnement intégré de développement et d'apprentissage de Python et constitue une alternative à la ligne de commande. Comme son nom l'indique, IDLE est très utile pour développer un nouveau code ou apprendre le python. Sous Windows, ceci est fourni avec l'interpréteur Python, mais dans d'autres systèmes d'exploitation, vous devrez peut-être l'installer via votre gestionnaire de paquets.

Les principaux objectifs de IDLE sont:

- Éditeur de texte multi-fenêtre avec mise en évidence de la syntaxe, auto-complétion et mise en retrait intelligente
- Shell Python avec coloration syntaxique
- Débogueur intégré avec étapes, points d'arrêt persistants et visibilité de la pile d'appels
- Indentation automatique (utile pour les débutants en apprentissage de l'indentation de Python)
- Sauvegardez le programme Python en tant que fichiers .py et exécutez-les et modifiez-les ultérieurement à l'aide d'IDLE.

Dans IDLE, appuyez sur `F5` ou `run Python Shell` pour lancer un interpréteur. Utiliser IDLE peut être une meilleure expérience d'apprentissage pour les nouveaux utilisateurs car le code est interprété comme l'utilisateur écrit.

Notez qu'il existe de nombreuses alternatives, voir par exemple cette discussion ou cette liste .

Dépannage

- **les fenêtres**

Si vous utilisez Windows, la commande par défaut est `python` . Si vous recevez une erreur `"'python' is not recognized"` , la cause la plus probable est que l'emplacement de Python ne se trouve pas dans la variable d'environnement `PATH` votre système. Vous pouvez y accéder en cliquant avec le bouton droit sur "Poste de travail" et en sélectionnant "Propriétés" ou en naviguant sur "Système" via "Panneau de configuration". Cliquez sur "Paramètres système avancés" puis sur "Variables d'environnement ...". Modifiez la variable `PATH` pour inclure le répertoire de votre installation Python, ainsi que le dossier Script (généralement `C:\Python27;C:\Python27\Scripts`). Cela nécessite des privilèges administratifs et peut nécessiter un redémarrage.

Lorsque vous utilisez plusieurs versions de Python sur le même `python.exe` , une solution possible consiste à renommer l'un des fichiers `python.exe` . Par exemple, en nommant une version `python27.exe` , `python27` deviendrait la commande Python pour cette version.

Vous pouvez également utiliser Python Launcher pour Windows, disponible via le programme d'installation et fourni par défaut. Il vous permet de sélectionner la version de Python à exécuter en utilisant `py -[xy]` au lieu de `python[xy]` . Vous pouvez utiliser la dernière version de Python 2 en exécutant des scripts avec `py -2` et la dernière version de Python 3 en exécutant des scripts avec `py -3` .

- **Debian / Ubuntu / MacOS**

  Cette section suppose que l'emplacement de l'exécutable `python` a été ajouté à la variable d'environnement `PATH` .

  Si vous êtes sur Debian / Ubuntu / MacOS, ouvrez le terminal et tapez `python` pour Python 2.x ou `python3` pour Python 3.x.

  Tapez le `which python` pour voir quel interpréteur Python sera utilisé.

- **Arch Linux**

  Python par défaut sur Arch Linux (et ses descendants) est Python 3, utilisez donc `python` ou `python3` pour Python 3.x et `python2` pour Python 2.x.

- **D'autres systèmes**

  Python 3 est parfois lié à `python` au lieu de `python3` . Pour utiliser Python 2 sur ces systèmes sur lesquels il est installé, vous pouvez utiliser `python2` .

Types de données

Types intégrés

Booléens

`bool` : Une valeur booléenne de `True` ou `False` . Les opérations logiques comme `and` `or` `not` peuvent

`not` être effectuées sur des booléens.

```
x or y    # if x is False then y otherwise x
x and y   # if x is False then x otherwise y
not x     # if x is True then False, otherwise True
```

Dans Python 2.x et dans Python 3.x, un booléen est aussi un `int` . Le type `bool` est une sous-classe du type `int` et `True` et `False` sont ses seules instances:

```
issubclass(bool, int) # True

isinstance(True, bool) # True
isinstance(False, bool) # True
```

Si des valeurs booléennes sont utilisées dans les opérations arithmétiques, leurs valeurs entières (`1` et `0` pour `True` et `False`) seront utilisées pour renvoyer un résultat entier:

```
True + False == 1 # 1 + 0 == 1
True * True  == 1 # 1 * 1 == 1
```

Nombres

- `int` : nombre entier

  ```
  a = 2
  b = 100
  c = 123456789
  d = 38563846326424324
  ```

  Les entiers en Python sont de tailles arbitraires.

  Remarque: dans les anciennes versions de Python, un type `long` était disponible, distinct de `int` . Les deux ont été unifiés.

- `float` : nombre à virgule flottante; la précision dépend de l'implémentation et de l'architecture du système, pour CPython, le type de données `float` correspond à un double C.

  ```
  a = 2.0
  b = 100.e0
  c = 123456789.e1
  ```

- `complex` : nombres complexes

  ```
  a = 2 + 1j
  b = 100 + 10j
  ```

Les opérateurs `<` , `<=` , `>` et `>=` `TypeError` une exception `TypeError` quand un opérande est un nombre complexe.

Cordes

Python 3.x 3.0

- `str` : une **chaîne Unicode** . Le type de `'hello'`
- `bytes` : une **chaîne d'octets** . Le type de `b'hello'`

Python 2.x 2.7

- `str` : une **chaîne d'octets** . Le type de `'hello'`
- `bytes` : synonyme de `str`
- `unicode` : une **chaîne Unicode** . Le type de `u'hello'`

Séquences et collections

Python différencie les séquences ordonnées et les collections non ordonnées (telles que `set` et `dict`).

- les chaînes (`str` , `bytes` , `unicode`) sont des séquences
- `reversed` : ordre inverse de `str` avec fonction `reversed`

```
a = reversed('hello')
```

- `tuple` : Collection ordonnée de `n` valeurs de tout type (`n >= 0`).

```
a = (1, 2, 3)
b = ('a', 1, 'python', (1, 2))
b[2] = 'something else' # returns a TypeError
```

Supporte l'indexation immuable; hashable si tous ses membres sont lavables

- `list` : Une collection ordonnée de `n` valeurs (`n >= 0`)

```
a = [1, 2, 3]
b = ['a', 1, 'python', (1, 2), [1, 2]]
b[2] = 'something else' # allowed
```

Non lavable; mutable.

- `set` : Une collection non ordonnée de valeurs uniques. Les articles doivent être lavables .

```
a = {1, 2, 'a'}
```

- `dict` : Une collection non ordonnée de paires clé-valeur uniques; les clés doivent être lavables .

```
a = {1: 'one',
```

```
     2: 'two'}

b = {'a': [1, 2, 3],
     'b': 'a string'}
```

Un objet est hashrable s'il a une valeur de hachage qui ne change jamais au cours de sa vie (il a besoin d'une `__hash__()`) et peut être comparé à d'autres objets (il nécessite une `__eq__()`). Les objets lavables qui comparent l'égalité doivent avoir la même valeur de hachage.

Constantes intégrées

En conjonction avec les types de données intégrés, il existe un petit nombre de constantes intégrées dans l'espace de noms intégré:

- `True` : La valeur réelle du type intégré `bool`
- `False` : La valeur false du type intégré `bool`
- `None` : objet singleton utilisé pour signaler qu'une valeur est absente.
- `Ellipsis` ou `...` : utilisé dans Python3 + de base n'importe où et utilisation limitée dans Python2.7 + dans le cadre de la notation de tableau. `numpy` et les paquets associés l'utilisent comme référence "include everything" dans les tableaux.
- `NotImplemented` : un singleton utilisé pour indiquer à Python qu'une méthode spéciale ne prend pas en charge les arguments spécifiques, et Python essaiera des alternatives si elles sont disponibles.

```
a = None # No value will be assigned. Any valid datatype can be assigned later
```

Python 3.x 3.0

`None` n'a aucun ordre naturel. L'utilisation des opérateurs de comparaison de commandes (`<` , `<=` , `>=` , `>`) n'est plus prise en charge et `TypeError` une `TypeError` .

Python 2.x 2.7

`None` n'est toujours inférieur à aucun nombre (`None < -32` `None` vaut `True`).

Test du type de variables

En python, nous pouvons vérifier le type de données d'un objet en utilisant le `type` fonction intégré.

```
a = '123'
print(type(a))
# Out: <class 'str'>
b = 123
print(type(b))
# Out: <class 'int'>
```

Dans les instructions conditionnelles, il est possible de tester le type de données avec `isinstance` . Cependant, il n'est généralement pas recommandé de compter sur le type de la variable.

```
i = 7
if isinstance(i, int):
    i += 1
elif isinstance(i, str):
    i = int(i)
    i += 1
```

Pour plus d'informations sur les différences entre `type()` et `isinstance()` lisez: Différences entre isinstance et type dans Python

Pour tester si quelque chose est de `NoneType` :

```
x = None
if x is None:
    print('Not a surprise, I just defined x as None.')
```

Conversion entre types de données

Vous pouvez effectuer une conversion de type de données explicite.

Par exemple, '123' est de type `str` et peut être converti en entier en utilisant la fonction `int` .

```
a = '123'
b = int(a)
```

La conversion à partir d'une chaîne de caractères telle que «123.456» peut être effectuée à l'aide de la fonction `float` .

```
a = '123.456'
b = float(a)
c = int(a)    # ValueError: invalid literal for int() with base 10: '123.456'
d = int(b)    # 123
```

Vous pouvez également convertir des types de séquence ou de collection

```
a = 'hello'
list(a)  # ['h', 'e', 'l', 'l', 'o']
set(a)   # {'o', 'e', 'l', 'h'}
tuple(a) # ('h', 'e', 'l', 'l', 'o')
```

Type de chaîne explicite à la définition des littéraux

Avec des étiquettes d'une lettre juste devant les guillemets, vous pouvez indiquer le type de

chaîne que vous souhaitez définir.

- `b'foo bar'` : `bytes` résultats dans Python 3, `str` dans Python 2
- `u'foo bar'` : résultats `str` dans Python 3, `unicode` dans Python 2
- `'foo bar'` : résultats `str`
- `r'foo bar'` : résultat dit chaîne brute, où échapper des caractères spéciaux n'est pas nécessaire, tout est pris textuellement à mesure que vous tapez

```
normal  = 'foo\nbar'   # foo
                       # bar
escaped = 'foo\\nbar'  # foo\nbar
raw     = r'foo\nbar'  # foo\nbar
```

Types de données mutables et immuables

Un objet est appelé *mutable* s'il peut être modifié. Par exemple, lorsque vous transmettez une liste à une fonction, la liste peut être modifiée:

```
def f(m):
    m.append(3)  # adds a number to the list. This is a mutation.

x = [1, 2]
f(x)
x == [1, 2]  # False now, since an item was added to the list
```

Un objet est appelé *immuable* s'il ne peut être modifié d'aucune façon. Par exemple, les entiers sont immuables, car il est impossible de les modifier:

```
def bar():
    x = (1, 2)
    g(x)
    x == (1, 2)  # Will always be True, since no function can change the object (1, 2)
```

Notez que les **variables** elles-mêmes sont mutables, nous pouvons donc réaffecter la *variable* `x` , mais cela ne change pas l'objet que `x` avait précédemment indiqué. Il ne fait `x` pointer vers un nouvel objet.

Les types de données dont les instances sont modifiables sont appelés *types de données mutables* , de même pour les objets et les types de données immuables.

Exemples de types de données immuables:

- `int` , `long` , `float` , `complex`
- `str`
- `bytes`
- `tuple`
- `frozenset`

Exemples de types de données mutables:

- `bytearray`
- `list`
- `set`
- `dict`

Modules et fonctions intégrés

Un module est un fichier contenant des définitions et des instructions Python. La fonction est un morceau de code qui exécute une logique.

```
>>> pow(2,3)    #8
```

Pour vérifier la fonction intégrée dans python, nous pouvons utiliser `dir()`. Si elle est appelée sans argument, retournez les noms dans la portée actuelle. Sinon, retournez une liste alphabétique des noms comprenant (certains) l'attribut de l'objet donné, et des attributs accessibles à partir de celui-ci.

```
>>> dir(__builtins__)
[
    'ArithmeticError',
    'AssertionError',
    'AttributeError',
    'BaseException',
    'BufferError',
    'BytesWarning',
    'DeprecationWarning',
    'EOFError',
    'Ellipsis',
    'EnvironmentError',
    'Exception',
    'False',
    'FloatingPointError',
    'FutureWarning',
    'GeneratorExit',
    'IOError',
    'ImportError',
    'ImportWarning',
    'IndentationError',
    'IndexError',
    'KeyError',
    'KeyboardInterrupt',
    'LookupError',
    'MemoryError',
    'NameError',
    'None',
    'NotImplemented',
    'NotImplementedError',
    'OSError',
    'OverflowError',
    'PendingDeprecationWarning',
    'ReferenceError',
    'RuntimeError',
    'RuntimeWarning',
    'StandardError',
    'StopIteration',
    'SyntaxError',
    'SyntaxWarning',
```

```
    'SystemError',
    'SystemExit',
    'TabError',
    'True',
    'TypeError',
    'UnboundLocalError',
    'UnicodeDecodeError',
    'UnicodeEncodeError',
    'UnicodeError',
    'UnicodeTranslateError',
    'UnicodeWarning',
    'UserWarning',
    'ValueError',
    'Warning',
    'ZeroDivisionError',
    '__debug__',
    '__doc__',
    '__import__',
    '__name__',
    '__package__',
    'abs',
    'all',
    'any',
    'apply',
    'basestring',
    'bin',
    'bool',
    'buffer',
    'bytearray',
    'bytes',
    'callable',
    'chr',
    'classmethod',
    'cmp',
    'coerce',
    'compile',
    'complex',
    'copyright',
    'credits',
    'delattr',
    'dict',
    'dir',
    'divmod',
    'enumerate',
    'eval',
    'execfile',
    'exit',
    'file',
    'filter',
    'float',
    'format',
    'frozenset',
    'getattr',
    'globals',
    'hasattr',
    'hash',
    'help',
    'hex',
    'id',
    'input',
    'int',
```

```
    'intern',
    'isinstance',
    'issubclass',
    'iter',
    'len',
    'license',
    'list',
    'locals',
    'long',
    'map',
    'max',
    'memoryview',
    'min',
    'next',
    'object',
    'oct',
    'open',
    'ord',
    'pow',
    'print',
    'property',
    'quit',
    'range',
    'raw_input',
    'reduce',
    'reload',
    'repr',
    'reversed',
    'round',
    'set',
    'setattr',
    'slice',
    'sorted',
    'staticmethod',
    'str',
    'sum',
    'super',
    'tuple',
    'type',
    'unichr',
    'unicode',
    'vars',
    'xrange',
    'zip'
]
```

Pour connaître la fonctionnalité de toute fonction, nous pouvons utiliser l' `help` fonction intégrée.

```
>>> help(max)
Help on built-in function max in module __builtin__:
max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
```

Les modules intégrés contiennent des fonctionnalités supplémentaires. Par exemple, pour obtenir la racine carrée d'un nombre, nous devons inclure un module `math` .

```
>>> import math
>>> math.sqrt(16) # 4.0
```

Pour connaître toutes les fonctions d'un module, nous pouvons affecter la liste des fonctions à une variable, puis imprimer la variable.

```
>>> import math
>>> dir(math)

   ['__doc__', '__name__', '__package__', 'acos', 'acosh',
   'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
   'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
   'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
   'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
   'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
   'tan', 'tanh', 'trunc']
```

il semble que `__doc__` soit utile pour fournir de la documentation dans, disons, des fonctions

```
>>> math.__doc__
'This module is always available.  It provides access to the\nmathematical
 functions defined by the C standard.'
```

Outre les fonctions, la documentation peut également être fournie sous forme de modules. Donc, si vous avez un fichier nommé `helloWorld.py` comme ceci:

```
"""This is the module docstring."""

def sayHello():
    """This is the function docstring."""
    return 'Hello World'
```

Vous pouvez accéder à ses docstrings comme ceci:

```
>>> import helloWorld
>>> helloWorld.__doc__
'This is the module docstring.'
>>> helloWorld.sayHello.__doc__
'This is the function docstring.'
```

- Pour tout type défini par l'utilisateur, ses attributs, ses attributs de classe et, de manière récursive, les attributs des classes de base de sa classe peuvent être récupérés à l'aide de dir ().

```
>>> class MyClassObject(object):
...     pass
...
>>> dir(MyClassObject)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Tout type de données peut être simplement converti en chaîne en utilisant une fonction intégrée

appelée `str` . Cette fonction est appelée par défaut lorsqu'un type de données est transmis à l' `print`

```
>>> str(123)    # "123"
```

Indentation de bloc

Python utilise l'indentation pour définir les constructions de contrôle et de boucle. Cela contribue à la lisibilité de Python, cependant, le programmeur doit faire très attention à l'utilisation des espaces. Ainsi, une erreur d'équilibrage de l'éditeur peut entraîner un comportement inattendu du code.

Python utilise le symbole de deux points (`:`) et en retrait pour montrer où des blocs de code début et la fin (Si vous venez d' une autre langue, ne pas confondre avec étant en quelque sorte lié à l' opérateur ternaire). En d'autres termes, les blocs en Python, tels que les fonctions, les boucles, les clauses `if` et les autres constructions, n'ont aucun identificateur de fin. Tous les blocs commencent par deux points et contiennent ensuite les lignes en retrait.

Par exemple:

```
def my_function():    # This is a function definition. Note the colon (:)
    a = 2             # This line belongs to the function because it's indented
    return a          # This line also belongs to the same function
print(my_function())  # This line is OUTSIDE the function block
```

ou

```
if a > b:             # If block starts here
    print(a)          # This is part of the if block
else:                 # else must be at the same level as if
    print(b)          # This line is part of the else block
```

Les blocs qui contiennent exactement une instruction sur une seule ligne peuvent être placés sur la même ligne, bien que cette forme ne soit généralement pas considérée comme un bon style:

```
if a > b: print(a)
else: print(b)
```

Essayer de le faire avec plus d'une seule déclaration *ne* fonctionnera *pas* :

```
if x > y: y = x
    print(y) # IndentationError: unexpected indent

if x > y: while y != z: y -= 1  # SyntaxError: invalid syntax
```

Un bloc vide provoque une `IndentationError` d' `IndentationError` . Utilisez `pass` (une commande qui ne fait rien) lorsque vous avez un bloc sans contenu:

```
def will_be_implemented_later():
    pass
```

Espaces vs. Onglets

En bref: utilisez **toujours** 4 espaces pour l'indentation.

L'utilisation exclusive des onglets est possible, mais PEP 8 , le guide de style pour le code Python, indique que les espaces sont préférés.

Python 3.x 3.0

Python 3 ne permet pas de combiner l'utilisation des tabulations et des espaces pour l'indentation. Dans ce cas, une erreur de compilation est générée: `Inconsistent use of tabs and spaces in indentation` et le programme ne s'exécutera pas.

Python 2.x 2.7

Python 2 permet de mélanger des tabulations et des espaces en indentation; ceci est fortement déconseillé. Le caractère de tabulation complète l'indentation précédente pour être un multiple de **8** espaces . Comme il est courant que les éditeurs soient configurés pour afficher des onglets comme multiples de **4** espaces, cela peut provoquer des bogues subtils.

Citant PEP 8 :

> Lorsque vous appelez l'interpréteur de ligne de commande Python 2 avec l'option `-t` , il émet des avertissements sur le code qui mélange illégalement des onglets et des espaces. Lors de l'utilisation de `-tt` ces avertissements deviennent des erreurs. Ces options sont fortement recommandées!

De nombreux éditeurs ont une configuration "tabs to spaces". Lors de la configuration de l'éditeur, il convient de différencier le *caractère de* tabulation ('\ t') et la touche `Tab` .

- Le *caractère de* tabulation doit être configuré pour afficher 8 espaces, correspondant à la sémantique du langage - au moins dans les cas où une indentation mixte (accidentelle) est possible. Les éditeurs peuvent également convertir automatiquement le caractère de tabulation en espaces.
- Cependant, il peut être utile de configurer l'éditeur de sorte que le fait d'appuyer sur la touche de `tabulation` insère 4 espaces, au lieu d'insérer un caractère de tabulation.

Le code source Python écrit avec un mélange de tabulations et d'espaces, ou avec un nombre d'espaces d'indentation non standard peut être rendu conforme à pep8 en utilisant autopep8 . (Une alternative moins puissante est fournie avec la plupart des installations Python: reindent.py)

Types de collection

Il existe un certain nombre de types de collection en Python. Bien que les types tels que `int` et `str` contiennent une seule valeur, les types de collection contiennent plusieurs valeurs.

Des listes

Le type de `list` est probablement le type de collection le plus utilisé en Python. Malgré son nom, une liste ressemble plus à un tableau dans d'autres langues, principalement du JavaScript. En Python, une liste est simplement une collection ordonnée de valeurs Python valides. Une liste peut être créée en entourant les valeurs, séparées par des virgules, entre crochets:

```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

Une liste peut être vide:

```
empty_list = []
```

Les éléments d'une liste ne sont pas limités à un seul type de données, ce qui est logique étant donné que Python est un langage dynamique:

```
mixed_list = [1, 'abc', True, 2.34, None]
```

Une liste peut contenir une autre liste comme élément:

```
nested_list = [['a', 'b', 'c'], [1, 2, 3]]
```

Les éléments d'une liste sont accessibles via un *index* ou une représentation numérique de leur position. Les listes de Python sont *indexées sur zéro,* ce qui signifie que le premier élément de la liste est à l'index 0, le deuxième élément à l'index 1 et ainsi de suite:

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
```

Les indices peuvent également être négatifs, ce qui signifie compter à partir de la fin de la liste (`-1` étant l'indice du dernier élément). Donc, en utilisant la liste de l'exemple ci-dessus:

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

Les listes sont modifiables, vous pouvez donc modifier les valeurs d'une liste:

```
names[0] = 'Ann'
print(names)
# Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

En outre, il est possible d'ajouter et / ou de supprimer des éléments d'une liste:

Ajouter un objet à la fin de la liste avec `L.append(object)` , renvoie `None` .

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# Outputs ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Ajouter un nouvel élément à répertorier à un index spécifique. `L.insert(index, object)`

```
names.insert(1, "Nikki")
print(names)
# Outputs ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Supprimer la première occurrence d'une valeur avec `L.remove(value)` , renvoie `None`

```
names.remove("Bob")
print(names) # Outputs ['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

Récupère l'index dans la liste du premier élément dont la valeur est x. Il affichera une erreur s'il n'y en a pas.

```
name.index("Alice")
0
```

Compter la longueur de la liste

```
len(names)
6
```

compter l'occurrence de n'importe quel élément de la liste

```
a = [1, 1, 1, 2, 3, 4]
a.count(1)
3
```

Inverser la liste

```
a.reverse()
[4, 3, 2, 1, 1, 1]
# or
a[::-1]
[4, 3, 2, 1, 1, 1]
```

Supprimer et renvoyer l'élément à l'index (par défaut au dernier élément) avec `L.pop([index])` , renvoie l'élément

```
names.pop() # Outputs 'Sia'
```

Vous pouvez parcourir les éléments de la liste comme ci-dessous:

```
for element in my_list:
    print (element)
```

Tuples

Un `tuple` est similaire à une liste sauf qu'il est de longueur fixe et immuable. Ainsi, les valeurs du

tuple ne peuvent pas être modifiées ni les valeurs ajoutées ou supprimées du tuple. Les tuples sont couramment utilisés pour les petites collections de valeurs qui n'auront pas besoin d'être modifiées, telles qu'une adresse IP et un port. Les tuples sont représentés par des parenthèses au lieu de crochets:

```
ip_address = ('10.20.30.40', 8080)
```

Les mêmes règles d'indexation pour les listes s'appliquent également aux tuples. Les tuples peuvent également être imbriqués et les valeurs peuvent être tout Python valide valide.

Un tuple avec un seul membre doit être défini (notez la virgule) de cette façon:

```
one_member_tuple = ('Only member',)
```

ou

```
one_member_tuple = 'Only member',   # No brackets
```

ou simplement en utilisant la syntaxe `tuple`

```
one_member_tuple = tuple(['Only member'])
```

Dictionnaires

Un `dictionary` en Python est une collection de paires clé-valeur. Le dictionnaire est entouré d'accolades. Chaque paire est séparée par une virgule et la clé et la valeur sont séparées par deux points. Voici un exemple:

```
state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

Pour obtenir une valeur, référez-vous à sa clé:

```
ca_capital = state_capitals['California']
```

Vous pouvez également obtenir toutes les clés dans un dictionnaire, puis les parcourir:

```
for k in state_capitals.keys():
    print('{} is the capital of {}'.format(state_capitals[k], k))
```

Les dictionnaires ressemblent fortement à la syntaxe JSON. Le module `json` natif de la bibliothèque standard Python peut être utilisé pour convertir entre JSON et les dictionnaires.

ensemble

Un `set` est une collection d'éléments sans répétitions et sans ordre d'insertion, mais triés. Ils sont utilisés dans des situations où il est important que certaines choses soient regroupées et non dans quel ordre. Pour les grands groupes de données, il est beaucoup plus rapide de vérifier si un élément se trouve dans un `set` ou de faire la même chose pour une `list` .

La définition d'un `set` est très similaire à la définition d'un `dictionary` :

```
first_names = {'Adam', 'Beth', 'Charlie'}
```

Ou vous pouvez créer un `set` utilisant une `list` existante:

```
my_list = [1,2,3]
my_set = set(my_list)
```

Vérifier l'appartenance à l' `set` utilisant `in` :

```
if name in first_names:
    print(name)
```

Vous pouvez effectuer une itération sur un `set` exactement comme une liste, mais rappelez-vous que les valeurs seront dans un ordre arbitraire défini par la mise en œuvre.

par défaut

Un `defaultdict` est un dictionnaire avec une valeur par défaut pour les clés, de sorte que les clés pour lesquelles aucune valeur n'a été explicitement définie sont accessibles sans erreur. `defaultdict` est particulièrement utile lorsque les valeurs du dictionnaire sont des collections (listes, dicts, etc.) dans le sens où il n'est pas nécessaire de l'initialiser chaque fois qu'une nouvelle clé est utilisée.

Un `defaultdict` ne `defaultdict` jamais une erreur de clé. Toute clé inexistante renvoie la valeur par défaut.

Par exemple, considérez le dictionnaire suivant

```
>>> state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

Si nous essayons d'accéder à une clé inexistante, python nous renvoie une erreur comme suit

```
>>> state_capitals['Alabama']
Traceback (most recent call last):

  File "<ipython-input-61-236329695e6f>", line 1, in <module>
    state_capitals['Alabama']

KeyError: 'Alabama'
```

Essayons avec un `defaultdict` par `defaultdict` . On peut le trouver dans le module des collections.

```
>>> from collections import defaultdict
>>> state_capitals = defaultdict(lambda: 'Boston')
```

Ce que nous avons fait ici est de définir une valeur par défaut (**Boston**) dans le cas où la clé de don n'existe pas. Maintenant, remplissez le dict comme avant:

```
>>> state_capitals['Arkansas'] = 'Little Rock'
>>> state_capitals['California'] = 'Sacramento'
>>> state_capitals['Colorado'] = 'Denver'
>>> state_capitals['Georgia'] = 'Atlanta'
```

Si nous essayons d'accéder au dict avec une clé inexistante, python nous renverra la valeur par défaut, à savoir Boston

```
>>> state_capitals['Alabama']
'Boston'
```

et renvoie les valeurs créées pour la clé existante, tout comme un `dictionary` normal

```
>>> state_capitals['Arkansas']
'Little Rock'
```

Utilitaire d'aide

Python a plusieurs fonctions intégrées dans l'interpréteur. Si vous souhaitez obtenir des informations sur les mots-clés, les fonctions intégrées, les modules ou les sujets, ouvrez une console Python et entrez:

```
>>> help()
```

Vous recevrez des informations en entrant directement les mots-clés:

```
>>> help(help)
```

ou dans l'utilitaire:

```
help> help
```

qui affichera une explication:

```
Help on _Helper in module _sitebuiltins object:

class _Helper(builtins.object)
 |  Define the builtin 'help'.
 |
 |  This is a wrapper around pydoc.help that provides a helpful message
 |  when 'help' is typed at the Python interactive prompt.
```

```
 |
 |  Calling help() at the Python prompt starts an interactive help session.
 |  Calling help(thing) prints help for the python object 'thing'.
 |
 |  Methods defined here:
 |
 |  __call__(self, *args, **kwds)
 |
 |  __repr__(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Vous pouvez également demander des sous-classes de modules:

```
help(pymysql.connections)
```

Vous pouvez utiliser l'aide pour accéder aux docstrings des différents modules que vous avez importés, par exemple, essayez ce qui suit:

```
>>> help(math)
```

et vous aurez une erreur

```
>>> import math
>>> help(math)
```

Et maintenant, vous obtiendrez une liste des méthodes disponibles dans le module, mais seulement après que vous l'ayez importé.

Fermez l'aide avec `quit`

Créer un module

Un module est un fichier importable contenant des définitions et des instructions.

Un module peut être créé en créant un fichier `.py` .

```
# hello.py
def say_hello():
    print("Hello!")
```

Les fonctions d'un module peuvent être utilisées en important le module.

Pour les modules que vous avez créés, ils devront être dans le même répertoire que le fichier dans lequel vous les importez. (Cependant, vous pouvez également les placer dans le répertoire

Python lib avec les modules pré-inclus, mais vous devriez les éviter si possible.)

```
$ python
>>> import hello
>>> hello.say_hello()
=> "Hello!"
```

Les modules peuvent être importés par d'autres modules.

```
# greet.py
import hello
hello.say_hello()
```

Des fonctions spécifiques d'un module peuvent être importées.

```
# greet.py
from hello import say_hello
say_hello()
```

Les modules peuvent être aliasés.

```
# greet.py
import hello as ai
ai.say_hello()
```

Un module peut être un script exécutable autonome.

```
# run_hello.py
if __name__ == '__main__':
    from hello import say_hello
    say_hello()
```

Exécuter!

```
$ python run_hello.py
=> "Hello!"
```

Si le module se trouve dans un répertoire et doit être détecté par python, le répertoire doit contenir un fichier nommé `__init__.py` .

Fonction de chaîne - str () et repr ()

Deux fonctions peuvent être utilisées pour obtenir une représentation lisible d'un objet.

`repr(x)` appelle `x.__repr__()` : une représentation de `x` . `eval` convertira généralement le résultat de cette fonction en objet d'origine.

`str(x)` appelle `x.__str__()` : une chaîne lisible par l'homme qui décrit l'objet. Cela peut éluder certains détails techniques.

---

repr ()

Pour de nombreux types, cette fonction tente de renvoyer une chaîne qui donnerait un objet ayant la même valeur lorsqu'il est transmis à `eval()` . Sinon, la représentation est une chaîne entre crochets qui contient le nom du type de l'objet ainsi que des informations supplémentaires. Cela inclut souvent le nom et l'adresse de l'objet.

str ()

Pour les chaînes, cela renvoie la chaîne elle-même. La différence entre ceci et `repr(object)` est que `str(object)` ne tente pas toujours de renvoyer une chaîne acceptable pour `eval()` . Son objectif est plutôt de retourner une chaîne imprimable ou lisible par l'homme. Si aucun argument n'est donné, cela retourne la chaîne vide, `''` .

Exemple 1:

```
s = """w'o"w"""
repr(s) # Output: '\'w\\\'o"w\''
str(s)  # Output: 'w\'o"w'
eval(str(s)) == s  # Gives a SyntaxError
eval(repr(s)) == s # Output: True
```

Exemple 2:

```
import datetime
today = datetime.datetime.now()
str(today)  # Output: '2016-09-15 06:58:46.915000'
repr(today) # Output: 'datetime.datetime(2016, 9, 15, 6, 58, 46, 915000)'
```

Lors de l'écriture d'une classe, vous pouvez remplacer ces méthodes pour faire ce que vous voulez:

```
class Represent(object):

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "Represent(x={},y=\"{}\")".format(self.x, self.y)

    def __str__(self):
        return "Representing x as {} and y as {}".format(self.x, self.y)
```

En utilisant la classe ci-dessus, nous pouvons voir les résultats:

```
r = Represent(1, "Hopper")
print(r)  # prints __str__
print(r.__repr__)  # prints __repr__: '<bound method Represent.__repr__ of
Represent(x=1,y="Hopper")>'
rep = r.__repr__()  # sets the execution of __repr__ to a new variable
print(rep)  # prints 'Represent(x=1,y="Hopper")'
```

```
r2 = eval(rep) # evaluates rep
print(r2)  # prints __str__ from new object
print(r2 == r)  # prints 'False' because they are different objects
```

Installation de modules externes à l'aide de pip

`pip` est votre ami lorsque vous devez installer un paquet à partir de la pléthore de choix disponibles dans l'index du package python (PyPI). `pip` est déjà installé si vous utilisez Python 2> = 2.7.9 ou Python 3> = 3.4 téléchargé depuis python.org. Pour les ordinateurs exécutant Linux ou un autre * nix avec un gestionnaire de paquets natif, `pip` doit souvent être installé manuellement.

Sur les instances sur `pip3` Python 2 et Python 3 sont installés, `pip` fait souvent référence à Python 2 et à `pip3` à Python 3. L'utilisation de `pip` n'installe que des packages pour Python 2 et `pip3` installe uniquement des packages pour Python 3.

Rechercher / installer un paquet

Rechercher un paquet est aussi simple que de taper

```
$ pip search <query>
# Searches for packages whose name or summary contains <query>
```

L'installation d'un paquet est aussi simple que de taper *(dans un terminal / une invite de commande, pas dans l'interpréteur Python)*

```
$ pip install [package_name]           # latest version of the package

$ pip install [package_name]==x.x.x    # specific version of the package

$ pip install '[package_name]>=x.x.x'  # minimum version of the package
```

où `xxx` est le numéro de version du package que vous souhaitez installer.

Lorsque votre serveur est derrière un proxy, vous pouvez installer le package en utilisant la commande ci-dessous:

```
$ pip --proxy http://<server address>:<port> install
```

Mise à niveau des packages installés

Lorsque de nouvelles versions de packages installés apparaissent, elles ne sont pas automatiquement installées sur votre système. Pour avoir un aperçu des paquets installés qui sont devenus obsolètes, exécutez:

```
$ pip list --outdated
```

Pour mettre à niveau un package spécifique

```
$ pip install [package_name] --upgrade
```

La mise à jour de tous les paquets obsolètes n'est pas une fonctionnalité standard de `pip` .

Mise à niveau pip

Vous pouvez mettre à niveau votre installation existante de pip en utilisant les commandes suivantes

- Sous Linux ou MacOS X:

  ```
  $ pip install -U pip
  ```

  Vous devrez peut-être utiliser `sudo` avec pip sur certains systèmes Linux

- Sous Windows:

  ```
  py -m pip install -U pip
  ```

  ou

  ```
  python -m pip install -U pip
  ```

Pour plus d'informations concernant pip, lisez ici .

Installation de Python 2.7.x et 3.x

> **Remarque** : Les instructions suivantes sont écrites pour Python 2.7 (sauf indication contraire): les instructions pour Python 3.x sont similaires.

LES FENÊTRES

Tout d'abord, téléchargez la dernière version de Python 2.7 à partir du site Web officiel (https://www.python.org/downloads/) . La version est fournie en tant que package MSI. Pour l'installer manuellement, double-cliquez simplement sur le fichier.

Par défaut, Python installe dans un répertoire:

```
C:\Python27\
```

Attention: l'installation ne modifie pas automatiquement la variable d'environnement PATH.

En supposant que votre installation Python se trouve dans C: \ Python27, ajoutez ceci à votre PATH:

```
C:\Python27\;C:\Python27\Scripts\
```

Maintenant, pour vérifier si l'installation de Python est valide, écrivez dans cmd:

```
python --version
```

Python 2.x et 3.x côte à côte

Pour installer et utiliser Python 2.x et 3.x côte à côte sur une machine Windows:

1. Installez Python 2.x à l'aide du programme d'installation MSI.

    - Assurez-vous que Python est installé pour tous les utilisateurs.
    - Facultatif: ajoutez Python à `PATH` pour pouvoir `PATH` Python 2.x à partir de la ligne de commande en utilisant `python` .

2. Installez Python 3.x en utilisant son installateur respectif.

    - Encore une fois, assurez-vous que Python est installé pour tous les utilisateurs.
    - Facultatif: ajoutez Python à `PATH` pour pouvoir `PATH` Python 3.x à partir de la ligne de commande en utilisant `python` . Cela peut remplacer `PATH` paramètres `PATH` Python 2.x. Vérifiez donc votre `PATH` et assurez-vous qu'il est configuré selon vos préférences.
    - Assurez-vous d'installer le `py launcher` pour tous les utilisateurs.

Python 3 va installer le lanceur Python qui peut être utilisé pour lancer Python 2.x et Python 3.x de manière interchangeable depuis la ligne de commande:

```
P:\>py -3
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

C:\>py -2
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Pour utiliser la version correspondante de `pip` pour une version Python spécifique, utilisez:

```
C:\>py -3 -m pip -V
pip 9.0.1 from C:\Python36\lib\site-packages (python 3.6)

C:\>py -2 -m pip -V
pip 9.0.1 from C:\Python27\lib\site-packages (python 2.7)
```

LINUX

Les dernières versions de CentOS, Fedora, Redhat Enterprise (RHEL) et Ubuntu sont livrées avec Python 2.7.

Pour installer Python 2.7 sur Linux manuellement, procédez comme suit dans le terminal:

```
wget --no-check-certificate https://www.python.org/ftp/python/2.7.X/Python-2.7.X.tgz
tar -xzf Python-2.7.X.tgz
cd Python-2.7.X
./configure
make
sudo make install
```

Ajoutez également le chemin du nouveau python dans la variable d'environnement PATH. Si le nouveau python est dans `/root/python-2.7.X` alors lancez `export PATH = $PATH:/root/python-2.7.X`

Maintenant, pour vérifier si l'installation de Python est valide en écriture dans le terminal:

```
python --version
```

Ubuntu (De la source)

Si vous avez besoin de Python 3.6, vous pouvez l'installer à partir du source, comme indiqué ci-dessous (Ubuntu 16.10 et 17.04 ont une version 3.6 dans le référentiel universel). Les étapes ci-dessous doivent être suivies pour Ubuntu 16.04 et les versions inférieures:

```
sudo apt install build-essential checkinstall
sudo apt install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev
libgdbm-dev libc6-dev libbz2-dev
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tar.xz
tar xvf Python-3.6.1.tar.xz
cd Python-3.6.1/
./configure --enable-optimizations
sudo make altinstall
```

macOS

En ce moment, macOS est installé avec Python 2.7.10, mais cette version est obsolète et légèrement modifiée par rapport au Python classique.

> La version de Python fournie avec OS X est idéale pour l'apprentissage, mais elle n'est pas bonne pour le développement. La version livrée avec OS X peut être obsolète par rapport à la version officielle actuelle de Python, considérée comme la version de production stable. (source)

Installez Homebrew :

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Installez Python 2.7:

```
brew install python
```

Pour Python 3.x, utilisez plutôt la commande `brew install python3` .

Lire Démarrer avec le langage Python en ligne:

https://riptutorial.com/fr/python/topic/193/demarrer-avec-le-langage-python

Chapitre 2: * args et ** kwargs

Remarques

Il y a quelques choses à noter:

1. Les noms `args` et `kwargs` sont utilisés par convention, ils ne font pas partie de la spécification du langage. Ainsi, ceux-ci sont équivalents:

```
def func(*args, **kwargs):
    print(args)
    print(kwargs)
```

```
def func(*a, **b):
    print(a)
    print(b)
```

2. Vous ne pouvez pas avoir plus d'un `args` ou plus d'un paramètre `kwargs` (toutefois, ils ne sont pas obligatoires)

```
def func(*args1, *args2):
#   File "<stdin>", line 1
#     def test(*args1, *args2):
#                      ^
# SyntaxError: invalid syntax
```

```
def test(**kwargs1, **kwargs2):
#   File "<stdin>", line 1
#     def test(**kwargs1, **kwargs2):
#                       ^
# SyntaxError: invalid syntax
```

3. Si un argument de position suit `*args` , ce sont des arguments de mots-clés uniquement pouvant être transmis uniquement par nom. Une seule étoile peut être utilisée à la place de `*args` pour forcer les valeurs à être des arguments de mot-clé sans fournir de liste de paramètres variadiques. Les listes de paramètres par mot clé uniquement sont uniquement disponibles dans Python 3.

```
def func(a, b, *args, x, y):
    print(a, b, args, x, y)

func(1, 2, 3, 4, x=5, y=6)
#>>> 1, 2, (3, 4), 5, 6
```

```
def func(a, b, *, x, y):
    print(a, b, x, y)

func(1, 2, x=5, y=6)
#>>> 1, 2, 5, 6
```

4. `**kwargs` doit venir en dernier dans la liste des paramètres.

```
def test(**kwargs, *args):
#   File "<stdin>", line 1
#     def test(**kwargs, *args):
#                        ^
# SyntaxError: invalid syntax
```

Examples

Utiliser * args lors de l'écriture de fonctions

Vous pouvez utiliser l'étoile * lors de l'écriture d'une fonction pour collecter tous les arguments de position (c'est-à-dire sans nom) dans un tuple:

```
def print_args(farg, *args):
   print("formal arg: %s" % farg)
   for arg in args:
       print("another positional arg: %s" % arg)
```

Méthode d'appel:

```
print_args(1, "two", 3)
```

Dans cet appel, farg sera affecté comme toujours, et les deux autres seront introduits dans le tuple args, dans l'ordre dans lequel ils ont été reçus.

Utiliser ** kwargs pour écrire des fonctions

Vous pouvez définir une fonction qui prend un nombre arbitraire d'arguments par mot-clé en utilisant l'étoile double `**` avant un nom de paramètre:

```
def print_kwargs(**kwargs):
    print(kwargs)
```

Lors de l'appel de la méthode, Python va construire un dictionnaire de tous les arguments de mots clés et le rendre disponible dans le corps de la fonction:

```
print_kwargs(a="two", b=3)
# prints: "{a: "two", b=3}"
```

Notez que le paramètre ** kwargs dans la définition de la fonction doit toujours être le dernier paramètre, et qu'il ne correspondra qu'aux arguments transmis après les précédents.

```
def example(a, **kw):
    print kw

example(a=2, b=3, c=4) # => {'b': 3, 'c': 4}
```

À l'intérieur du corps de la fonction, `kwargs` est manipulé de la même manière qu'un dictionnaire; pour accéder à des éléments individuels dans `kwargs` vous suffit de les parcourir comme vous le feriez avec un dictionnaire normal:

```
def print_kwargs(**kwargs):
    for key in kwargs:
        print("key = {0}, value = {1}".format(key, kwargs[key]))
```

Maintenant, appeler `print_kwargs(a="two", b=1)` affiche la sortie suivante:

```
print_kwargs(a = "two", b = 1)
key = a, value = "two"
key = b, value = 1
```

Utiliser * args lors de l'appel de fonctions

Un cas d'utilisation courant pour `*args` dans une définition de fonction consiste à déléguer le traitement à une fonction encapsulée ou héritée. Un exemple typique pourrait être la méthode `__init__` une classe

```
class A(object):
    def __init__(self, b, c):
        self.y = b
        self.z = c

class B(A):
    def __init__(self, a, *args, **kwargs):
        super(B, self).__init__(*args, **kwargs)
        self.x = a
```

Ici, le `a` paramètre est traité par la classe enfant après que tous les autres arguments (position et mot - clé) sont passés sur - et traitées par - la classe de base.

Par exemple:

```
b = B(1, 2, 3)
b.x  # 1
b.y  # 2
b.z  # 3
```

Ce qui se passe ici, c'est que la fonction `__init__` la classe `B` voit les arguments `1, 2, 3` . Il sait qu'il doit prendre un argument positionnel (`a`), donc il saisit le premier argument passé dans (`1`), donc dans le cadre de la fonction `a == 1` .

Ensuite, il voit qu'il doit prendre un nombre arbitraire d'arguments positionnels (`*args`), il prend donc le reste des arguments positionnels passés dans (`1, 2`) et les place dans `*args` . Maintenant

(dans le cadre de la fonction) `args == [2, 3]` .

Ensuite, il appelle la fonction `__init__` classe `A` avec `*args` . Python voit le `*` devant args et "décompresse" la liste en arguments. Dans cet exemple, lorsque la classe `B` 's `__init__` fonction appelle la classe `A` ' s `__init__` fonction, il sera transmis les arguments `2, 3` (c. -à- `A(2, 3)`).

Enfin, il définit sa propre propriété `x` sur le premier argument positionnel `a` , qui est égal à `1` .

Utiliser ** kwargs lors de l'appel de fonctions

Vous pouvez utiliser un dictionnaire pour attribuer des valeurs aux paramètres de la fonction. en utilisant des paramètres name comme clés dans le dictionnaire et la valeur de ces arguments liés à chaque clé:

```
def test_func(arg1, arg2, arg3): # Usual function with three arguments
   print("arg1: %s" % arg1)
   print("arg2: %s" % arg2)
   print("arg3: %s" % arg3)

# Note that dictionaries are unordered, so we can switch arg2 and arg3. Only the names matter.
kwargs = {"arg3": 3, "arg2": "two"}

# Bind the first argument (ie. arg1) to 1, and use the kwargs dictionary to bind the others
test_var_args_call(1, **kwargs)
```

Utiliser * args lors de l'appel de fonctions

L'utilisation de l'opérateur `*` sur un argument lors de l'appel d'une fonction consiste à décompresser la liste ou un argument tuple

```
def print_args(arg1, arg2):
    print(str(arg1) + str(arg2))

a = [1,2]
b = tuple([3,4])

print_args(*a)
# 12
print_args(*b)
# 34
```

Notez que la longueur de l'argument étoilé doit être égale au nombre d'arguments de la fonction.

Un idiome commun de python est d'utiliser l'opérateur de décompression `*` avec la fonction `zip` pour inverser ses effets:

```
a = [1,3,5,7,9]
b = [2,4,6,8,10]

zipped = zip(a,b)
# [(1,2), (3,4), (5,6), (7,8), (9,10)]

zip(*zipped)
```

```
# (1,3,5,7,9), (2,4,6,8,10)
```

Arguments relatifs aux mots clés uniquement et aux mots clés

Python 3 vous permet de définir des arguments de fonction qui ne peuvent être assignés que par mot-clé, même sans valeurs par défaut. Cela se fait en utilisant star * pour consommer des paramètres de position supplémentaires sans définir les paramètres du mot-clé. Tous les arguments après le * sont des arguments uniquement par mot clé (c'est-à-dire non positionnels). Notez que si les arguments contenant uniquement des mots clés ne sont pas définis par défaut, ils sont toujours requis lors de l'appel de la fonction.

```
def print_args(arg1, *args, keyword_required, keyword_only=True):
    print("first positional arg: {}".format(arg1))
    for arg in args:
        print("another positional arg: {}".format(arg))
    print("keyword_required value: {}".format(keyword_required))
    print("keyword_only value: {}".format(keyword_only))

print(1, 2, 3, 4) # TypeError: print_args() missing 1 required keyword-only argument:
'keyword_required'
print(1, 2, 3, keyword_required=4)
# first positional arg: 1
# another positional arg: 2
# another positional arg: 3
# keyword_required value: 4
# keyword_only value: True
```

Remplir les valeurs de kwarg avec un dictionnaire

```
def foobar(foo=None, bar=None):
    return "{}{}".format(foo, bar)

values = {"foo": "foo", "bar": "bar"}

foobar(**values) # "foobar"
```

** kwargs et valeurs par défaut

Utiliser les valeurs par défaut avec ** kwargs

```
def fun(**kwargs):
    print kwargs.get('value', 0)

fun()
# print 0
fun(value=1)
# print 1
```

Lire * args et ** kwargs en ligne: https://riptutorial.com/fr/python/topic/2475/--args-et----kwargs

Chapitre 3: Accéder au code source Python et au bytecode

Examples

Affiche le bytecode d'une fonction

L'interpréteur Python compile le code en bytecode avant de l'exécuter sur la machine virtuelle Python (voir aussi What is python bytecode?.

Voici comment afficher le bytecode d'une fonction Python

```
import dis

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)

# Display the disassembled bytecode of the function.
dis.dis(fib)
```

La fonction `dis.dis` du module dis retournera un bytecode décompilé de la fonction qui lui est transmise.

Explorer l'objet code d'une fonction

CPython permet d'accéder à l'objet code pour un objet fonction.

L'objet `__code__` contient le bytecode brut (`co_code`) de la fonction ainsi que d'autres informations telles que les constantes et les noms de variables.

```
def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)
```

Affiche le code source d'un objet

Objets qui ne sont pas intégrés

Pour imprimer le code source d'un objet Python, utilisez `inspect` . Notez que cela ne fonctionnera pas pour les objets intégrés ni pour les objets définis de manière interactive. Pour cela, vous aurez besoin d'autres méthodes expliquées plus tard.

Voici comment imprimer le code source de la méthode `randint` partir du module `random` :

```
import random
import inspect

print(inspect.getsource(random.randint))
# Output:
#    def randint(self, a, b):
#        """Return random integer in range [a, b], including both end points.
#        """
#
#        return self.randrange(a, b+1)
```

Pour imprimer la chaîne de documentation

```
print(inspect.getdoc(random.randint))
# Output:
# Return random integer in range [a, b], including both end points.
```

Imprimer le chemin complet du fichier où la méthode `random.randint` est définie:

```
print(inspect.getfile(random.randint))
# c:\Python35\lib\random.py
print(random.randint.__code__.co_filename) # equivalent to the above
# c:\Python35\lib\random.py
```

Objets définis interactivement

Si un objet est défini de manière interactive, l' `inspect` ne peut pas fournir le code source, mais vous pouvez utiliser `dill.source.getsource` place.

```
# define a new function in the interactive shell
def add(a, b):
   return a + b
print(add.__code__.co_filename) # Output: <stdin>

import dill
print dill.source.getsource(add)
# def add(a, b):
      return a + b
```

Objets intégrés

Le code source des fonctions intégrées de Python est écrit en **c** et accessible uniquement en consultant le code source de Python (hébergé sur Mercurial ou téléchargeable sur https://www.python.org/downloads/source/) .

```
print(inspect.getsource(sorted)) # raises a TypeError
type(sorted) # <class 'builtin_function_or_method'>
```

Lire Accéder au code source Python et au bytecode en ligne:

https://riptutorial.com/fr/python/topic/4351/acceder-au-code-source-python-et-au-bytecode

Chapitre 4: Accès à la base de données

Remarques

Python peut gérer différents types de bases de données. Pour chacun de ces types, une API différente existe. Donc, encouragez la similitude entre ces différentes API, le PEP 249 a été introduit.

> Cette API a été définie pour encourager la similarité entre les modules Python utilisés pour accéder aux bases de données. Ce faisant, nous espérons obtenir une cohérence permettant de mieux comprendre les modules, du code généralement plus portable entre les bases de données et une plus grande portée de la connectivité aux bases de données à partir de Python. PEP-249

Examples

Accéder à la base de données MySQL en utilisant MySQLdb

La première chose à faire est de créer une connexion à la base de données en utilisant la méthode connect. Après cela, vous aurez besoin d'un curseur qui fonctionnera avec cette connexion.

Utilisez la méthode execute du curseur pour interagir avec la base de données et, de temps en temps, validez les modifications à l'aide de la méthode de validation de l'objet de connexion.

Une fois que tout est fait, n'oubliez pas de fermer le curseur et la connexion.

Voici une classe Dbconnect avec tout ce dont vous aurez besoin.

```
import MySQLdb

class Dbconnect(object):

    def __init__(self):

        self.dbconection = MySQLdb.connect(host='host_example',
                                           port=int('port_example'),
                                           user='user_example',
                                           passwd='pass_example',
                                           db='schema_example')
        self.dbcursor = self.dbconection.cursor()

    def commit_db(self):
        self.dbconection.commit()

    def close_db(self):
        self.dbcursor.close()
        self.dbconection.close()
```

L'interaction avec la base de données est simple. Après avoir créé l'objet, utilisez simplement la

méthode execute.

```
db = Dbconnect()
db.dbcursor.execute('SELECT * FROM %s' % 'table_example')
```

Si vous souhaitez appeler une procédure stockée, utilisez la syntaxe suivante. Notez que la liste des paramètres est facultative.

```
db = Dbconnect()
db.callproc('stored_procedure_name', [parameters] )
```

Une fois la requête terminée, vous pouvez accéder aux résultats de plusieurs manières. L'objet curseur est un générateur qui peut récupérer tous les résultats ou être en boucle.

```
results = db.dbcursor.fetchall()
for individual_row in results:
    first_field = individual_row[0]
```

Si vous voulez une boucle utilisant directement le générateur:

```
for individual_row in db.dbcursor:
    first_field = individual_row[0]
```

Si vous souhaitez valider les modifications apportées à la base de données:

```
db.commit_db()
```

Si vous voulez fermer le curseur et la connexion:

```
db.close_db()
```

SQLite

SQLite est une base de données légère basée sur disque. Comme il ne nécessite pas de serveur de base de données séparé, il est souvent utilisé pour le prototypage ou pour les petites applications souvent utilisées par un seul utilisateur ou par un utilisateur à un moment donné.

```
import sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

c.execute("CREATE TABLE user (name text, age integer)")

c.execute("INSERT INTO user VALUES ('User A', 42)")
c.execute("INSERT INTO user VALUES ('User B', 43)")

conn.commit()

c.execute("SELECT * FROM user")
```

```
print(c.fetchall())

conn.close()
```

Le code ci-dessus se connecte à la base de données stockée dans le fichier nommée `users.db` , en créant d'abord le fichier s'il n'existe pas déjà. Vous pouvez interagir avec la base de données via des instructions SQL.

Le résultat de cet exemple devrait être:

```
[(u'User A', 42), (u'User B', 43)]
```

La syntaxe SQLite: une analyse approfondie

Commencer

1. Importez le module sqlite en utilisant

   ```
   >>> import sqlite3
   ```

2. Pour utiliser le module, vous devez d'abord créer un objet Connection qui représente la base de données. Ici, les données seront stockées dans le fichier example.db:

   ```
   >>> conn = sqlite3.connect('users.db')
   ```

   Alternativement, vous pouvez également fournir le nom spécial `:memory:` pour créer une base de données temporaire en RAM, comme suit:

   ```
   >>> conn = sqlite3.connect(':memory:')
   ```

3. Une fois que vous avez une `Connection` , vous pouvez créer un objet `Cursor` et appeler sa méthode `execute()` pour exécuter les commandes SQL:

   ```
   c = conn.cursor()

   # Create table
   c.execute('''CREATE TABLE stocks
                (date text, trans text, symbol text, qty real, price real)''')

   # Insert a row of data
   c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

   # Save (commit) the changes
   conn.commit()

   # We can also close the connection if we are done with it.
   # Just be sure any changes have been committed or they will be lost.
   conn.close()
   ```

Attributs importants et fonctions de `Connection`

1. `isolation_level`

   C'est un attribut utilisé pour obtenir ou définir le niveau d'isolation actuel. Aucun pour le mode autocommit ou l'un des modes `DEFERRED` , `IMMEDIATE` ou `EXCLUSIVE` .

2. `cursor`

   L'objet curseur est utilisé pour exécuter des commandes et des requêtes SQL.

3. `commit()`

   Valide la transaction en cours.

4. `rollback()`

   Annule les modifications apportées depuis l'appel précédent à `commit()`

5. `close()`

   Ferme la connexion à la base de données. Il n'appelle pas `commit()` automatiquement. Si `close()` est appelée sans d'abord appeler `commit()` (en supposant que vous n'êtes pas en mode autocommit), toutes les modifications apportées seront perdues.

6. `total_changes`

   Un attribut qui enregistre le nombre total de lignes modifiées, supprimées ou insérées depuis l'ouverture de la base de données.

7. `execute` , `executemany` et `executescript`

   Ces fonctions fonctionnent de la même manière que celles de l'objet curseur. Ceci est un raccourci car l'appel de ces fonctions via l'objet de connexion entraîne la création d'un objet curseur intermédiaire et appelle la méthode correspondante de l'objet curseur

8. `row_factory`

   Vous pouvez remplacer cet attribut par un appelable qui accepte le curseur et la ligne d'origine sous la forme d'un tuple et renverra la ligne de résultat réel.

```
def dict_factory(cursor, row):
    d = {}
    for i, col in enumerate(cursor.description):
        d[col[0]] = row[i]
    return d

conn = sqlite3.connect(":memory:")
conn.row_factory = dict_factory
```

Fonctions importantes du `Cursor`

1. `execute(sql[, parameters])`

Exécute une *seule* instruction SQL. L'instruction SQL peut être paramétrée (c.-à-d. Des espaces réservés au lieu de littéraux SQL). Le module sqlite3 prend en charge deux types d'espaces réservés: les points d'interrogation `?` («Style qmark») et les espaces réservés nommés `:name` («style nommé»).

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.execute("create table people (name, age)")

who = "Sophia"
age = 37
# This is the qmark style:
cur.execute("insert into people values (?, ?)",
            (who, age))

# And this is the named style:
cur.execute("select * from people where name=:who and age=:age",
            {"who": who, "age": age})  # the keys correspond to the placeholders in SQL

print(cur.fetchone())
```

Attention: n'utilisez pas `%s` pour insérer des chaînes dans les commandes SQL, car cela peut rendre votre programme vulnérable à une attaque par injection SQL (voir Injection SQL).

2. `executemany(sql, seq_of_parameters)`

   Exécute une commande SQL contre toutes les séquences de paramètres ou mappages trouvés dans la séquence sql. Le module sqlite3 permet également d'utiliser un itérateur générant des paramètres au lieu d'une séquence.

```
L = [(1, 'abcd', 'dfj', 300),    # A list of tuples to be inserted into the database
     (2, 'cfgd', 'dyfj', 400),
     (3, 'sdd', 'dfjh', 300.50)]

conn = sqlite3.connect("test1.db")
conn.execute("create table if not exists book (id int, name text, author text, price
real)")
conn.executemany("insert into book values (?, ?, ?, ?)", L)

for row in conn.execute("select * from book"):
    print(row)
```

   Vous pouvez également transmettre des objets d'itérateur en tant que paramètre à executemany, et la fonction effectuera une itération sur chaque tuple de valeurs renvoyé par l'itérateur. L'itérateur doit retourner un tuple de valeurs.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
```

```
        return self

    def __next__(self):            # (use next(self) for Python 2)
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),)

conn = sqlite3.connect("abc.db")
cur = conn.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

rows = cur.execute("select c from characters")
for row in rows:
    print(row[0]),
```

3. `executescript(sql_script)`

   Ceci est une méthode de commodité non standard pour exécuter plusieurs instructions SQL à la fois. Il émet d'abord une instruction `COMMIT` , puis exécute le script SQL obtenu en tant que paramètre.

   `sql_script` peut être une instance de `str` ou `bytes` .

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.executescript("""
     create table person(
         firstname,
         lastname,
         age
     );

     create table book(
         title,
         author,
         published
     );

     insert into book(title, author, published)
     values (
         'Dirk Gently''s Holistic Detective Agency',
         'Douglas Adams',
         1987
     );
     """)
```

Le prochain ensemble de fonctions est utilisé conjointement avec les `SELECT` en SQL. Pour récupérer des données après avoir exécuté une `SELECT` , vous pouvez traiter le curseur comme un itérateur, appeler la méthode `fetchone()` du curseur pour extraire une seule ligne correspondante ou appeler `fetchall()` pour obtenir la liste des lignes correspondantes.

Exemple de forme d'itérateur:

```
import sqlite3
stocks = [('2006-01-05', 'BUY', 'RHAT', 100, 35.14),
          ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.0),
          ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
conn = sqlite3.connect(":memory:")
conn.execute("create table stocks (date text, buysell text, symb text, amount int, price
real)")
conn.executemany("insert into stocks values (?, ?, ?, ?, ?)", stocks)
cur = conn.cursor()

for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
    print(row)

# Output:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

4. `fetchone()`

   Récupère la ligne suivante d'un ensemble de résultats de requête, en retournant une seule séquence ou Aucun lorsque aucune autre donnée n'est disponible.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
i = cur.fetchone()
while(i):
    print(i)
    i = cur.fetchone()

# Output:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

5. `fetchmany(size=cursor.arraysize)`

   Récupère le prochain ensemble de lignes d'un résultat de requête (spécifié par taille), renvoyant une liste. Si la taille est omise, fetchmany renvoie une seule ligne. Une liste vide est renvoyée lorsqu'aucune ligne supplémentaire n'est disponible.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchmany(2))

# Output:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)]
```

6. `fetchall()`

   Récupère toutes les lignes (restantes) d'un résultat de requête, renvoyant une liste.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchall())
```

```
# Output:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
('2006-04-06', 'SELL', 'IBM', 500, 53.0), ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
```

Types de données SQLite et Python

SQLite supporte nativement les types suivants: NULL, INTEGER, REAL, TEXT, BLOB.

Voici comment les types de données sont convertis lorsque vous passez de SQL à Python ou vice versa.

```
None     <->     NULL
int      <->     INTEGER/INT
float    <->     REAL/FLOAT
str      <->     TEXT/VARCHAR(n)
bytes    <->     BLOB
```

Accès à la base de données PostgreSQL avec psycopg2

psycopg2 est l'adaptateur de base de données PostgreSQL le plus populaire, à la fois léger et efficace. C'est l'implémentation actuelle de l'adaptateur PostgreSQL.

> Ses principales caractéristiques sont l'implémentation complète de la spécification Python DB API 2.0 et la sécurité des threads (plusieurs threads peuvent partager la même connexion)

Établir une connexion à la base de données et créer une table

```
import psycopg2

# Establish a connection to the database.
# Replace parameter values with database credentials.
conn = psycopg2.connect(database="testpython",
                        user="postgres",
                        host="localhost",
                        password="abc123",
                        port="5432")

# Create a cursor. The cursor allows you to execute database queries.
cur = conn.cursor()

# Create a table. Initialise the table name, the column names and data type.
cur.execute("""CREATE TABLE FRUITS (
                    id          INT ,
                    fruit_name  TEXT,
                    color       TEXT,
                    price       REAL
            )""")
conn.commit()
conn.close()
```

Insérer des données dans la table:

```
# After creating the table as shown above, insert values into it.
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Apples', 'green', 1.00)""")

cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Bananas', 'yellow', 0.80)""")
```

Récupération des données de la table:

```
# Set up a query and execute it
cur.execute("""SELECT id, fruit_name, color, price
             FROM fruits""")

# Fetch the data
rows = cur.fetchall()

# Do stuff with the data
for row in rows:
    print "ID = {} ".format(row[0])
    print "FRUIT NAME = {}".format(row[1])
    print("COLOR = {}".format(row[2]))
    print("PRICE = {}".format(row[3]))
```

Le résultat de ce qui précède serait:

```
ID = 1
NAME = Apples
COLOR = green
PRICE = 1.0

ID = 2
NAME = Bananas
COLOR = yellow
PRICE = 0.8
```

Et voilà, vous savez maintenant la moitié de tout ce que vous devez savoir sur ***psycopg2*** ! :)

Base de données Oracle

Conditions préalables:

- Package cx_Oracle - Voir ici pour toutes les versions
- Client instantané Oracle - Pour Windows x64 , Linux x64

Installer:

- Installez le package cx_Oracle en tant que:

  ```
  sudo rpm -i <YOUR_PACKAGE_FILENAME>
  ```

- Extrayez le client instantané Oracle et définissez les variables d'environnement comme suit:

```
ORACLE_HOME=<PATH_TO_INSTANTCLIENT>
PATH=$ORACLE_HOME:$PATH
LD_LIBRARY_PATH=<PATH_TO_INSTANTCLIENT>:$LD_LIBRARY_PATH
```

Créer une connexion:

```
import cx_Oracle

class OraExec(object):
    _db_connection = None
    _db_cur = None

    def __init__(self):
        self._db_connection =
            cx_Oracle.connect('<USERNAME>/<PASSWORD>@<HOSTNAME>:<PORT>/<SERVICE_NAME>')
        self._db_cur = self._db_connection.cursor()
```

Obtenir la version de la base de données:

```
ver = con.version.split(".")
print ver
```

Sortie d'échantillon: ['12', '1', '0', '2', '0']

Exécuter la requête: SELECT

```
_db_cur.execute("select * from employees order by emp_id")
for result in _db_cur:
    print result
```

La sortie sera en tuples Python:

(10, 'SYSADMIN', 'IT-INFRA', 7)

(23, 'HR ASSOCIATE', 'HUMAN RESOURCES', 6)

Exécuter la requête: INSERT

```
_db_cur.execute("insert into employees(emp_id, title, dept, grade)
                values (31, 'MTS', 'ENGINEERING', 7)
_db_connection.commit()
```

Lorsque vous effectuez des opérations d'insertion / mise à jour / suppression dans une base de données Oracle, les modifications sont uniquement disponibles dans votre session jusqu'à ce que la `commit` soit émise. Lorsque les données mises à jour sont validées dans la base de données, elles sont alors disponibles pour les autres utilisateurs et sessions.

Execute query: INSERT en utilisant les variables Bind

Référence

Les variables de liaison vous permettent de réexécuter des instructions avec de nouvelles valeurs, sans avoir à ré-analyser l'analyse. Les variables de liaison améliorent la réutilisation du code et peuvent réduire le risque d'attaques par injection SQL.

```
rows = [ (1, "First" ),
     (2, "Second" ),
     (3, "Third" ) ]
_db_cur.bindarraysize = 3
_db_cur.setinputsizes(int, 10)
_db_cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)
_db_connection.commit()
```

Fermer la connexion:

```
_db_connection.close()
```

La méthode close () ferme la connexion. Toute connexion non explicitement fermée sera automatiquement libérée à la fin du script.

Connexion

Créer une connexion

Selon PEP 249, la connexion à une base de données doit être établie à l'aide d'un constructeur `connect()` , qui renvoie un objet `Connection` . Les arguments de ce constructeur dépendent de la base de données. Reportez-vous aux rubriques spécifiques à la base de données pour connaître les arguments pertinents.

```
import MyDBAPI

con = MyDBAPI.connect(*database_dependent_args)
```

Cet objet de connexion a quatre méthodes:

1: fermer

```
con.close()
```

Ferme la connexion instantanément. Notez que la connexion est automatiquement fermée si la méthode `Connection.__del___` est appelée. Toute transaction en attente sera implicitement annulée.

2: commettre

```
con.commit()
```

Valide toute transaction en attente dans la base de données.

3: rollback

```
con.rollback()
```

Revient au début de toute transaction en attente. En d'autres termes, cela annule toute transaction non engagée dans la base de données.

4: curseur

```
cur = con.cursor()
```

Renvoie un objet `Cursor` . Ceci est utilisé pour effectuer des transactions sur la base de données.

Utiliser sqlalchemy

Pour utiliser sqlalchemy pour la base de données:

```
from sqlalchemy import create_engine
from sqlalchemy.engine.url import URL


url = URL(drivername='mysql',
          username='user',
          password='passwd',
          host='host',
          database='db')

engine = create_engine(url)  # sqlalchemy engine
```

Maintenant, ce moteur peut être utilisé: par exemple avec des pandas pour récupérer des dataframes directement depuis mysql

```
import pandas as pd

con = engine.connect()
dataframe = pd.read_sql(sql=query, con=con)
```

Lire Accès à la base de données en ligne: https://riptutorial.com/fr/python/topic/4240/acces-a-la-base-de-donnees

Chapitre 5: Accès aux attributs

Syntaxe

- `x.title # Accesses the title attribute using the dot notation`
- `x.title = "Hello World" # Sets the property of the title attribute using the dot notation`
- `@property # Used as a decorator before the getter method for properties`
- `@title.setter # Used as a decorator before the setter method for properties`

Examples

Accès aux attributs de base à l'aide de la notation par points

Prenons un exemple de classe.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

book1 = Book(title="Right Ho, Jeeves", author="P.G. Wodehouse")
```

En Python, vous pouvez accéder au *titre* de l'attribut de la classe en utilisant la notation par points.

```
>>> book1.title
'P.G. Wodehouse'
```

Si un attribut n'existe pas, Python génère une erreur:

```
>>> book1.series
Traceback  (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Book' object has no attribute 'series'
```

Setters, Getters & Properties

Pour des raisons d'encapsulation de données, vous voulez parfois avoir un attribut dont la valeur provient d'autres attributs ou, en général, quelle valeur doit être calculée pour le moment. La méthode standard pour gérer cette situation consiste à créer une méthode, appelée getter ou un setter.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

Dans l'exemple ci-dessus, il est facile de voir ce qui se passe si nous créons un nouveau livre contenant un titre et un auteur. Si tous les livres à ajouter à notre bibliothèque ont des auteurs et

des titres, nous pouvons ignorer les règles et les règles et utiliser la notation par points. Cependant, supposons que nous ayons des livres qui n'ont pas d'auteur et que nous voulons définir l'auteur sur "Unknown". Ou s'ils ont plusieurs auteurs et nous prévoyons de retourner une liste d'auteurs.

Dans ce cas, nous pouvons créer un getter et un setter pour l'attribut *author* .

```
class P:
    def __init__(self,title,author):
        self.title = title
        self.setAuthor(author)

    def get_author(self):
        return self.author

    def set_author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Ce schéma n'est pas recommandé.

Une des raisons est qu'il y a un piège: Supposons que nous ayons conçu notre classe avec l'attribut public et aucune méthode. Les gens l'ont déjà beaucoup utilisé et ils ont écrit un code comme celui-ci:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
```

Maintenant, nous avons un problème. Parce que l' *auteur* n'est pas un attribut! Python offre une solution à ce problème appelé propriétés. Une méthode pour obtenir les propriétés est décorée avec la propriété @ avant son en-tête. La méthode que nous voulons utiliser en tant que setter est décorée avec @ attributeName.setter avant elle.

Gardant cela à l'esprit, nous avons maintenant notre nouvelle classe mise à jour.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    @property
    def author(self):
        return self.__author

    @author.setter
    def author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Notez que normalement Python ne vous permet pas d'avoir plusieurs méthodes avec le même

nom et le même nombre de paramètres. Cependant, dans ce cas, Python le permet à cause des décorateurs utilisés.

Si nous testons le code:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
>>> book.author
Unknown
```

Lire Accès aux attributs en ligne: https://riptutorial.com/fr/python/topic/4392/acces-aux-attributs

Chapitre 6: Alternatives à changer de déclaration à partir d'autres langues

Remarques

Il n'y a *AUCUNE* instruction switch en python comme choix de conception de langage. Il y a eu un PEP (PEP-3103) couvrant le sujet qui a été rejeté.

Vous pouvez trouver de nombreuses listes de recettes sur la façon de faire vos propres instructions de commutation en python, et j'essaie ici de suggérer les options les plus sensées. Voici quelques endroits à vérifier:

- http://stackoverflow.com/questions/60208/replacements-for-switch-statement-in-python
- http://code.activestate.com/recipes/269708-some-python-style-switches/
- http://code.activestate.com/recipes/410692-readable-switch-construction-without-lambdas-or-di/
- …

Examples

Utilisez ce que le langage offre: la construction if / else.

Eh bien, si vous voulez une construction `switch` / `case` , la méthode la plus simple consiste à utiliser le bon vieux `if` / `else` :

```
def switch(value):
    if value == 1:
        return "one"
    if value == 2:
        return "two"
    if value == 42:
        return "the answer to the question about life, the universe and everything"
    raise Exception("No case found!")
```

il peut sembler redondant, et pas toujours joli, mais c'est de loin le moyen le plus efficace, et il fait le travail:

```
>>> switch(1)
one
>>> switch(2)
two
>>> switch(3)
…
Exception: No case found!
>>> switch(42)
the answer to the question about life the universe and everything
```

Utilisez un dicton de fonctions

Un autre moyen simple consiste à créer un dictionnaire de fonctions:

```
switch = {
    1: lambda: 'one',
    2: lambda: 'two',
    42: lambda: 'the answer of life the universe and everything',
}
```

puis vous ajoutez une fonction par défaut:

```
def default_case():
    raise Exception('No case found!')
```

et vous utilisez la méthode get du dictionnaire pour obtenir la fonction donnée la valeur pour vérifier et l'exécuter. Si value n'existe pas dans le dictionnaire, `default_case` est exécuté.

```
>>> switch.get(1, default_case)()
one
>>> switch.get(2, default_case)()
two
>>> switch.get(3, default_case)()
…
Exception: No case found!
>>> switch.get(42, default_case)()
the answer of life the universe and everything
```

vous pouvez aussi faire du sucre syntaxique pour que le commutateur soit plus joli:

```
def run_switch(value):
    return switch.get(value, default_case)()

>>> run_switch(1)
one
```

Utiliser l'introspection de classe

Vous pouvez utiliser une classe pour imiter la structure de commutateur / cas. Ce qui suit utilise l'introspection d'une classe (en utilisant la fonction `getattr()` qui résout une chaîne dans une méthode liée sur une instance) pour résoudre la partie "case".

Ensuite, cette méthode d'introspection est associée à la méthode `__call__` pour surcharger l'opérateur `()` .

```
class SwitchBase:
    def switch(self, case):
        m = getattr(self, 'case_{}'.format(case), None)
        if not m:
            return self.default
        return m
```

```
    __call__ = switch
```

Ensuite, pour le rendre plus joli, nous sous- `SwitchBase` classe `SwitchBase` (mais cela pourrait être fait dans une classe), et nous définissons ici tous les `case` comme des méthodes:

```
class CustomSwitcher:
    def case_1(self):
        return 'one'

    def case_2(self):
        return 'two'

    def case_42(self):
        return 'the answer of life, the universe and everything!'

    def default(self):
        raise Exception('Not a case!')
```

alors nous pouvons enfin l'utiliser:

```
>>> switch = CustomSwitcher()
>>> print(switch(1))
one
>>> print(switch(2))
two
>>> print(switch(3))
…
Exception: Not a case!
>>> print(switch(42))
the answer of life, the universe and everything!
```

Utiliser un gestionnaire de contexte

Une autre manière, très lisible et élégante, mais beaucoup moins efficace qu'une structure if / else, est de construire une classe comme suit, qui lira et stockera la valeur à comparer, s'exposera dans le contexte comme un appelable retournera true s'il correspond à la valeur stockée:

```
class Switch:
    def __init__(self, value):
        self._val = value
    def __enter__(self):
        return self
    def __exit__(self, type, value, traceback):
        return False # Allows traceback to occur
    def __call__(self, cond, *mconds):
        return self._val in (cond,)+mconds
```

alors la définition des cas correspond presque à la construction réelle du `switch` / de la `case` (exposée dans une fonction ci-dessous, pour faciliter la démonstration):

```
def run_switch(value):
    with Switch(value) as case:
```

```
        if case(1):
            return 'one'
        if case(2):
            return 'two'
        if case(3):
            return 'the answer to the question about life, the universe and everything'
        # default
        raise Exception('Not a case!')
```

Donc, l'exécution serait:

```
>>> run_switch(1)
one
>>> run_switch(2)
two
>>> run_switch(3)
…
Exception: Not a case!
>>> run_switch(42)
the answer to the question about life, the universe and everything
```

Nota Bene :

- Cette solution est proposée comme module de *commutation* disponible sur pypi .

Lire Alternatives à changer de déclaration à partir d'autres langues en ligne: https://riptutorial.com/fr/python/topic/4268/alternatives-a-changer-de-declaration-a-partir-d-autres-langues

Chapitre 7: Analyse des arguments de ligne de commande

Introduction

La plupart des outils de ligne de commande reposent sur des arguments transmis au programme lors de son exécution. Au lieu de demander une entrée, ces programmes attendent que les données ou les indicateurs spécifiques (qui deviennent des booléens) soient définis. Cela permet à l'utilisateur et aux autres programmes d'exécuter le fichier Python en lui transmettant les données au démarrage. Cette section explique et démontre l'implémentation et l'utilisation des arguments de ligne de commande dans Python.

Examples

Bonjour tout le monde en argparse

Le programme suivant dit bonjour à l'utilisateur. Il prend un argument positionnel, le nom de l'utilisateur, et peut également être dit le message d'accueil.

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('name',
    help='name of user'
)

parser.add_argument('-g', '--greeting',
    default='Hello',
    help='optional alternate greeting'
)

args = parser.parse_args()

print("{greeting}, {name}!".format(
       greeting=args.greeting,
       name=args.name)
)
```

```
$ python hello.py --help
usage: hello.py [-h] [-g GREETING] name

positional arguments:
  name                  name of user

optional arguments:
  -h, --help            show this help message and exit
  -g GREETING, --greeting GREETING
                        optional alternate greeting
```

```
$ python hello.py world
Hello, world!
$ python hello.py John -g Howdy
Howdy, John!
```

Pour plus de détails, veuillez lire la documentation d'argparse .

Exemple basique avec docopt

docopt tourne l'argument d'analyse de ligne de commande sur sa tête. Au lieu d'analyser les arguments, il vous suffit d' **écrire la chaîne d'utilisation** de votre programme, et docopt **analyse la chaîne d'utilisation** et l'utilise pour extraire les arguments de la ligne de commande.

```
"""
Usage:
    script_name.py [-a] [-b] <path>

Options:
    -a            Print all the things.
    -b            Get more bees into the path.
"""
from docopt import docopt


if __name__ == "__main__":
    args = docopt(__doc__)
    import pprint; pprint.pprint(args)
```

Échantillon exécute:

```
$ python script_name.py
Usage:
    script_name.py [-a] [-b] <path>
$ python script_name.py something
{'-a': False,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py something -a
{'-a': True,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py -b something -a
{'-a': True,
 '-b': True,
 '<path>': 'something'}
```

Définir des arguments mutuellement exclusifs avec argparse

Si vous souhaitez que deux ou plusieurs arguments soient mutuellement exclusifs. Vous pouvez utiliser la fonction `argparse.ArgumentParser.add_mutually_exclusive_group()` . Dans l'exemple ci-dessous, foo ou bar peut exister mais pas les deux en même temps.

```
import argparse
```

```
parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-f", "--foo")
group.add_argument("-b", "--bar")
args = parser.parse_args()
print "foo = ", args.foo
print "bar = ", args.bar
```

Si vous essayez d'exécuter le script en spécifiant à la fois les arguments `--foo` et `--bar` , le script se plaindra du message ci-dessous.

```
error: argument -b/--bar: not allowed with argument -f/--foo
```

Utilisation d'arguments en ligne de commande avec argv

Chaque fois qu'un script Python est appelé à partir de la ligne de commande, l'utilisateur peut fournir des **arguments de ligne de commande** supplémentaires qui seront transmis au script. Ces arguments seront disponibles pour le programmeur de la variable système `sys.argv` (« argv » est un nom traditionnel utilisé dans la plupart des langages de programmation, et cela signifie « **arg** ument **v** ecteur »).

Par convention, le premier élément de la liste `sys.argv` est le nom du script Python lui-même, tandis que le reste des éléments sont les jetons transmis par l'utilisateur lors de l'appel du script.

```
# cli.py
import sys
print(sys.argv)

$ python cli.py
=> ['cli.py']

$ python cli.py fizz
=> ['cli.py', 'fizz']

$ python cli.py fizz buzz
=> ['cli.py', 'fizz', 'buzz']
```

Voici un autre exemple d'utilisation de `argv` . Nous enlevons d'abord l'élément initial de sys.argv car il contient le nom du script. Ensuite, nous combinons le reste des arguments en une seule phrase et imprimons finalement cette phrase en préfixant le nom de l'utilisateur actuellement connecté (pour qu'il émule un programme de discussion).

```
import getpass
import sys

words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

L'algorithme couramment utilisé lors de l'analyse "manuelle" d'un certain nombre d'arguments non positionnels consiste à parcourir la liste `sys.argv` . L'une des façons consiste à parcourir la liste et à en afficher chaque élément:

```
# reverse and copy sys.argv
argv = reversed(sys.argv)
# extract the first element
arg = argv.pop()
# stop iterating when there's no more args to pop()
while len(argv) > 0:
    if arg in ('-f', '--foo'):
        print('seen foo!')
    elif arg in ('-b', '--bar'):
        print('seen bar!')
    elif arg in ('-a', '--with-arg'):
        arg = arg.pop()
        print('seen value: {}'.format(arg))
    # get the next value
    arg = argv.pop()
```

Message d'erreur d'analyseur personnalisé avec argparse

Vous pouvez créer des messages d'erreur d'analyseur en fonction des besoins de votre script. Ceci est à travers la fonction `argparse.ArgumentParser.error` . L'exemple ci-dessous montre le script imprimant une utilisation et un message d'erreur à `stderr` lorsque `--foo` est donné mais pas `--bar` .

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-f", "--foo")
parser.add_argument("-b", "--bar")
args = parser.parse_args()
if args.foo and args.bar is None:
    parser.error("--foo requires --bar. You did not specify bar.")

print "foo =", args.foo
print "bar =", args.bar
```

En supposant que votre nom de script est sample.py, et que nous `python sample.py --foo ds_in_fridge` : `python sample.py --foo ds_in_fridge`

Le script se plaindra de ce qui suit:

```
usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.
```

Groupement conceptuel d'arguments avec argparse.add_argument_group ()

Lorsque vous créez un argumentse ArgumentParser () et exécutez votre programme avec '-h', vous obtenez un message d'utilisation automatisé expliquant avec quels arguments vous pouvez exécuter votre logiciel. Par défaut, les arguments positionnels et les arguments conditionnels sont séparés en deux catégories. Par exemple, voici un petit script (example.py) et la sortie lorsque vous exécutez `python example.py -h` .

```
import argparse

parser = argparse.ArgumentParser(description='Simple example')
```

```
parser.add_argument('name', help='Who to greet', default='World')
parser.add_argument('--bar_this')
parser.add_argument('--bar_that')
parser.add_argument('--foo_this')
parser.add_argument('--foo_that')
args = parser.parse_args()
```

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                      [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                      name

Simple example

positional arguments:
  name                  Who to greet

optional arguments:
  -h, --help            show this help message and exit
  --bar_this BAR_THIS
  --bar_that BAR_THAT
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

Dans certaines situations, vous souhaitez séparer vos arguments en d'autres sections conceptuelles pour aider votre utilisateur. Par exemple, vous pouvez souhaiter avoir toutes les options de saisie dans un groupe et toutes les options de format de sortie dans un autre. L'exemple ci-dessus peut être ajusté pour séparer les `--foo_*` des `--bar_*` comme ceci.

```
import argparse

parser = argparse.ArgumentParser(description='Simple example')
parser.add_argument('name', help='Who to greet', default='World')
# Create two argument groups
foo_group = parser.add_argument_group(title='Foo options')
bar_group = parser.add_argument_group(title='Bar options')
# Add arguments to those groups
foo_group.add_argument('--bar_this')
foo_group.add_argument('--bar_that')
bar_group.add_argument('--foo_this')
bar_group.add_argument('--foo_that')
args = parser.parse_args()
```

Qui produit cette sortie lorsque `python example.py -h` est exécuté:

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                      [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                      name

Simple example

positional arguments:
  name                  Who to greet

optional arguments:
  -h, --help            show this help message and exit

Foo options:
```

```
  --bar_this BAR_THIS
  --bar_that BAR_THAT

Bar options:
  --foo_this FOO_THIS
  --foo_that FOO_THAT
```

Exemple avancé avec docopt et docopt_dispatch

Comme avec docopt, avec [docopt_dispatch] vous créez votre `--help` dans la variable `__doc__` de votre module de point d'entrée. Là, vous appelez `dispatch` avec la chaîne de caractères en argument, afin de pouvoir exécuter l'analyseur dessus.

Cela étant, au lieu de manipuler manuellement les arguments (qui se terminent généralement par une structure if / else cyclomatique élevée), vous laissez le soin d'expédier uniquement en fonction de la manière dont vous souhaitez gérer l'ensemble des arguments.

C'est à cela que sert le décorateur `dispatch.on` : vous lui donnez l'argument ou la séquence d'arguments qui devrait déclencher la fonction, et cette fonction sera exécutée avec les valeurs correspondantes en tant que paramètres.

```
"""Run something in development or production mode.

Usage: run.py --development <host> <port>
       run.py --production <host> <port>
       run.py items add <item>
       run.py items delete <item>

"""
from docopt_dispatch import dispatch

@dispatch.on('--development')
def development(host, port, **kwargs):
    print('in *development* mode')

@dispatch.on('--production')
def development(host, port, **kwargs):
    print('in *production* mode')

@dispatch.on('items', 'add')
def items_add(item, **kwargs):
    print('adding item...')

@dispatch.on('items', 'delete')
def items_delete(item, **kwargs):
    print('deleting item...')

if __name__ == '__main__':
    dispatch(__doc__)
```

Lire Analyse des arguments de ligne de commande en ligne:
https://riptutorial.com/fr/python/topic/1382/analyse-des-arguments-de-ligne-de-commande

Chapitre 8: Analyse HTML

Examples

Localiser un texte après un élément dans BeautifulSoup

Imaginez que vous avez le code HTML suivant:

```
<div>
    <label>Name:</label>
    John Smith
</div>
```

Et vous devez localiser le texte "John Smith" après l'élément `label` .

Dans ce cas, vous pouvez localiser l'élément `label` par texte, puis utiliser la propriété `.next_sibling` :

```
from bs4 import BeautifulSoup

data = """
<div>
    <label>Name:</label>
    John Smith
</div>
"""

soup = BeautifulSoup(data, "html.parser")

label = soup.find("label", text="Name:")
print(label.next_sibling.strip())
```

Imprime `John Smith` .

Utilisation de sélecteurs CSS dans BeautifulSoup

BeautifulSoup a un support limité pour les sélecteurs CSS , mais couvre les plus couramment utilisés. Utilisez la méthode `select()` pour rechercher plusieurs éléments et `select_one()` pour rechercher un seul élément.

Exemple de base:

```
from bs4 import BeautifulSoup

data = """
<ul>
    <li class="item">item1</li>
    <li class="item">item2</li>
    <li class="item">item3</li>
</ul>
"""
```

```
soup = BeautifulSoup(data, "html.parser")

for item in soup.select("li.item"):
    print(item.get_text())
```

Impressions:

```
item1
item2
item3
```

PyQuery

pyquery est une bibliothèque de type jquery pour python. Il supporte très bien les sélecteurs CSS.

```
from pyquery import PyQuery

html = """
<h1>Sales</h1>
<table id="table">
<tr>
    <td>Lorem</td>
    <td>46</td>
</tr>
<tr>
    <td>Ipsum</td>
    <td>12</td>
</tr>
<tr>
    <td>Dolor</td>
    <td>27</td>
</tr>
<tr>
    <td>Sit</td>
    <td>90</td>
</tr>
</table>
"""

doc = PyQuery(html)

title = doc('h1').text()

print title

table_data = []

rows = doc('#table > tr')
for row in rows:
    name = PyQuery(row).find('td').eq(0).text()
    value = PyQuery(row).find('td').eq(1).text()

    print "%s\t  %s" % (name, value)
```

Lire Analyse HTML en ligne: https://riptutorial.com/fr/python/topic/1384/analyse-html

Chapitre 9: Anti-Patterns Python

Examples

Trop zélé sauf clause

Les exceptions sont puissantes, mais une seule clause trop zélée peut éliminer tout cela en une seule ligne.

```
try:
    res = get_result()
    res = res[0]
    log('got result: %r' % res)
except:
    if not res:
        res = ''
    print('got exception')
```

Cet exemple montre 3 symptômes de l'anti-motif:

1. L' `except` sans type d'exception (ligne 5) attraperez exceptions même en bonne santé, y compris `KeyboardInterrupt` . Cela empêchera le programme de sortir dans certains cas.
2. Le bloc d'exception ne relance pas l'erreur, ce qui signifie que nous ne pourrons pas dire si l'exception provient de `get_result` ou parce que `res` était une liste vide.
3. Pire encore, si nous craignions que le résultat soit vide, nous avons causé quelque chose de bien pire. Si `get_result` échoue, `res` restera complètement désactivé et la référence à `res` dans le bloc except, augmentera `NameError` , masquant complètement l'erreur d'origine.

Pensez toujours au type d'exception que vous essayez de gérer. Donnez une lecture à la page des exceptions et obtenez une idée des exceptions de base.

Voici une version fixe de l'exemple ci-dessus:

```
import traceback

try:
    res = get_result()
except Exception:
    log_exception(traceback.format_exc())
    raise
try:
    res = res[0]
except IndexError:
    res = ''

log('got result: %r' % res)
```

Nous attrapons des exceptions plus spécifiques, en les relançant si nécessaire. Quelques lignes de plus, mais infiniment plus correctes.

Avant de sauter avec une fonction gourmande en processeurs

Un programme peut facilement perdre du temps en appelant plusieurs fois une fonction gourmande en processeurs.

Par exemple, prenez une fonction qui ressemble à ceci: elle retourne un entier si la `value` entrée peut en produire une, sinon `None` :

```
def intensive_f(value): # int -> Optional[int]
   # complex, and time-consuming code
   if process_has_failed:
       return None
   return integer_output
```

Et il pourrait être utilisé de la manière suivante:

```
x = 5
if intensive_f(x) is not None:
    print(intensive_f(x) / 2)
else:
    print(x, "could not be processed")

print(x)
```

Bien que cela fonctionne, il a pour problème d'appeler `intensive_f` , ce qui double la durée d'exécution du code. Une meilleure solution serait d'obtenir la valeur de retour de la fonction au préalable.

```
x = 5
result = intensive_f(x)
if result is not None:
    print(result / 2)
else:
    print(x, "could not be processed")
```

Cependant, une manière plus claire et peut - être plus pythonique consiste à utiliser des exceptions, par exemple:

```
x = 5
try:
    print(intensive_f(x) / 2)
except TypeError: # The exception raised if None + 1 is attempted
    print(x, "could not be processed")
```

Ici, aucune variable temporaire n'est nécessaire. Il est souvent préférable d'utiliser une instruction `assert` et d'attraper l' `AssertionError` place.

Clés de dictionnaire

Un exemple courant de l'endroit où cela se trouve est l'accès aux clés du dictionnaire. Par exemple comparer:

```
bird_speeds = get_very_long_dictionary()

if "european swallow" in bird_speeds:
    speed = bird_speeds["european swallow"]
else:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

avec:

```
bird_speeds = get_very_long_dictionary()

try:
    speed = bird_speeds["european swallow"]
except KeyError:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

Le premier exemple doit parcourir le dictionnaire deux fois, et comme il s'agit d'un long dictionnaire, cela peut prendre beaucoup de temps à chaque fois. La seconde ne nécessite qu'une seule recherche dans le dictionnaire, ce qui économise beaucoup de temps processeur.

Une alternative à ceci est d'utiliser `dict.get(key, default)` *, cependant de nombreuses circonstances peuvent nécessiter des opérations plus complexes dans le cas où la clé n'est pas présente.*

Lire Anti-Patterns Python en ligne: https://riptutorial.com/fr/python/topic/4700/anti-patterns-python

Chapitre 10: Appelez Python depuis C

Introduction

La documentation fournit un exemple d'implémentation de la communication inter-processus entre les scripts C # et Python.

Remarques

Notez que dans l'exemple ci-dessus, les données sont sérialisées en utilisant la bibliothèque **MongoDB.Bson** qui peut être installée via le gestionnaire NuGet.

Sinon, vous pouvez utiliser n'importe quelle bibliothèque de sérialisation JSON de votre choix.

Vous trouverez ci-dessous les étapes de mise en œuvre de la communication inter-processus:

- Les arguments en entrée sont sérialisés en chaîne JSON et enregistrés dans un fichier texte temporaire:

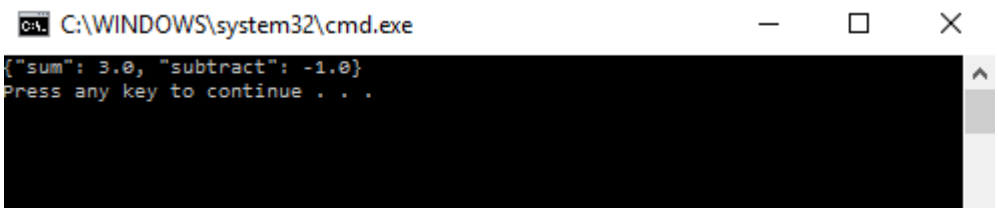
```
 BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");
 string argsFile = string.Format("{0}\\{1}.txt", Path.GetDirectoryName(pyScriptPath),
Guid.NewGuid());
```

- L'interpréteur Python python.exe exécute le script python qui lit la chaîne JSON à partir d'un fichier texte temporaire et des arguments d'entrée inversés:

```
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
   input_args = json.loads( data_file.read() )

x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]
```

- Le script Python est exécuté et le dictionnaire de sortie est sérialisé en chaîne JSON et imprimé dans la fenêtre de commande:

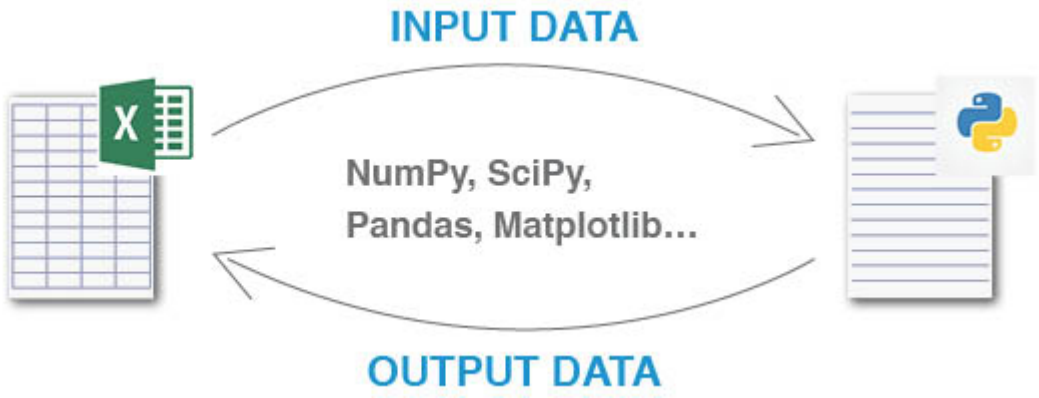
```
print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```



- Lire la chaîne JSON de sortie depuis l'application C #:

```
using (StreamReader myStreamReader = process.StandardOutput)
```

```
{
   outputString = myStreamReader.ReadLine();
   process.WaitForExit();
}
```



J'utilise la communication inter-processus entre les scripts C # et Python dans l'un de mes projets qui permet d'appeler des scripts Python directement à partir de feuilles de calcul Excel.

Le projet utilise le complément ExcelDNA pour la liaison C # - Excel.

Le code source est stocké dans le référentiel GitHub.

Vous trouverez ci-dessous des liens vers des pages wiki qui donnent un aperçu du projet et vous aident à démarrer en 4 étapes simples .

- Commencer
- Vue d'ensemble de la mise en œuvre
- Exemples
- Assistant d'objet
- Les fonctions

J'espère que vous trouvez l'exemple et le projet utiles.

Examples

Script Python à appeler par application C

```
import sys
import json

# load input arguments from the text file
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
    input_args = json.loads( data_file.read() )

# cast strings to floats
x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]
```

```
print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```

Code C # appelant le script Python

```
using MongoDB.Bson;
using System;
using System.Diagnostics;
using System.IO;

namespace python_csharp
{
    class Program
    {
        static void Main(string[] args)
        {
            // full path to .py file
            string pyScriptPath = "...../sum.py";
            // convert input arguments to JSON string
            BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");

            bool saveInputFile = false;

            string argsFile = string.Format("{0}\\{1}.txt",
Path.GetDirectoryName(pyScriptPath), Guid.NewGuid());

            string outputString = null;
            // create new process start info
            ProcessStartInfo prcStartInfo = new ProcessStartInfo
            {
                // full path of the Python interpreter 'python.exe'
                FileName = "python.exe", // string.Format(@"""{0}""", "python.exe"),
                UseShellExecute = false,
                RedirectStandardOutput = true,
                CreateNoWindow = false
            };

            try
            {
                // write input arguments to .txt file
                using (StreamWriter sw = new StreamWriter(argsFile))
                {
                    sw.WriteLine(argsBson);
                    prcStartInfo.Arguments = string.Format("{0} {1}",
string.Format(@"""{0}""", pyScriptPath), string.Format(@"""{0}""", argsFile));
                }
                // start process
                using (Process process = Process.Start(prcStartInfo))
                {
                    // read standard output JSON string
                    using (StreamReader myStreamReader = process.StandardOutput)
                    {
                        outputString = myStreamReader.ReadLine();
                        process.WaitForExit();
                    }
                }
            }
            finally
            {
                // delete/save temporary .txt file
```

```
                if (!saveInputFile)
                {
                    File.Delete(argsFile);
                }
            }
            Console.WriteLine(outputString);
        }
    }
}
```

Lire Appelez Python depuis C # en ligne: https://riptutorial.com/fr/python/topic/10759/appelez-python-depuis-c-sharp

Chapitre 11: Arbre de syntaxe abstraite

Examples

Analyser les fonctions dans un script python

Cela analyse un script python et, pour chaque fonction définie, indique le numéro de la ligne où la fonction a commencé, où la signature se termine, où se termine la définition de la fonction et où se termine la définition de la fonction.

```
#!/usr/local/bin/python3

import ast
import sys

""" The data we collect.  Each key is a function name; each value is a dict
with keys: firstline, sigend, docend, and lastline and values of line numbers
where that happens. """
functions = {}

def process(functions):
    """ Handle the function data stored in functions. """
    for funcname,data in functions.items():
        print("function:",funcname)
        print("\tstarts at line:",data['firstline'])
        print("\tsignature ends at line:",data['sigend'])
        if ( data['sigend'] < data['docend'] ):
            print("\tdocstring ends at line:",data['docend'])
        else:
            print("\tno docstring")
        print("\tfunction ends at line:",data['lastline'])
        print()

class FuncLister(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        """ Recursively visit all functions, determining where each function
        starts, where its signature ends, where the docstring ends, and where
        the function ends. """
        functions[node.name] = {'firstline':node.lineno}
        sigend = max(node.lineno,lastline(node.args))
        functions[node.name]['sigend'] = sigend
        docstring = ast.get_docstring(node)
        docstringlength = len(docstring.split('\n')) if docstring else -1
        functions[node.name]['docend'] = sigend+docstringlength
        functions[node.name]['lastline'] = lastline(node)
        self.generic_visit(node)

def lastline(node):
    """ Recursively find the last line of a node """
    return max( [ node.lineno if hasattr(node,'lineno') else -1 , ]
                +[lastline(child) for child in ast.iter_child_nodes(node)] )

def readin(pythonfilename):
    """ Read the file name and store the function data into functions. """
    with open(pythonfilename) as f:
        code = f.read()
```

```
    FuncLister().visit(ast.parse(code))

def analyze(file,process):
    """ Read the file and process the function data. """
    readin(file)
    process(functions)

if __name__ == '__main__':
    if len(sys.argv)>1:
        for file in sys.argv[1:]:
            analyze(file,process)
    else:
        analyze(sys.argv[0],process)
```

Lire Arbre de syntaxe abstraite en ligne: https://riptutorial.com/fr/python/topic/5370/arbre-de-syntaxe-abstraite

Chapitre 12: ArcPy

Remarques

Cet exemple utilise un curseur de recherche du module Data Access (da) de ArcPy.

Ne confondez pas la syntaxe arcpy.da.SearchCursor avec l'arcpy.SearchCursor () plus précoce et plus lent.

Le module d'accès aux données (arcpy.da) est uniquement disponible depuis ArcGIS 10.1 for Desktop.

Examples

Impression de la valeur d'un champ pour toutes les lignes de la classe d'entités dans la géodatabase fichier à l'aide du curseur de recherche

Pour imprimer un champ de test (TestField) à partir d'une classe d'entités de test (TestFC) dans une géodatabase de fichier de test (Test.gdb) située dans un dossier temporaire (C: \ Temp):

```
with arcpy.da.SearchCursor(r"C:\Temp\Test.gdb\TestFC",["TestField"]) as cursor:
    for row in cursor:
        print row[0]
```

createDissolvedGDB pour créer un fichier gdb sur l'espace de travail

```
def createDissolvedGDB(workspace, gdbName):
    gdb_name = workspace + "/" + gdbName + ".gdb"

    if(arcpy.Exists(gdb_name):
        arcpy.Delete_management(gdb_name)
        arcpy.CreateFileGDB_management(workspace, gdbName, "")
    else:
        arcpy.CreateFileGDB_management(workspace, gdbName, "")

    return gdb_name
```

Lire ArcPy en ligne: https://riptutorial.com/fr/python/topic/4693/arcpy

Chapitre 13: Augmenter les erreurs / exceptions personnalisées

Introduction

Python a de nombreuses exceptions intégrées qui forcent votre programme à générer une erreur lorsque quelque chose ne va pas.

Cependant, vous devrez parfois créer des exceptions personnalisées répondant à vos besoins.

En Python, les utilisateurs peuvent définir de telles exceptions en créant une nouvelle classe. Cette classe d'exception doit être dérivée, directement ou indirectement, de la classe Exception. La plupart des exceptions intégrées proviennent également de cette classe.

Examples

Exception personnalisée

Ici, nous avons créé une exception définie par l'utilisateur appelée CustomError qui est dérivée de la classe Exception. Cette nouvelle exception peut être déclenchée, comme d'autres exceptions, en utilisant l'instruction raise avec un message d'erreur facultatif.

```
class CustomError(Exception):
       pass

x = 1

if x == 1:
    raise CustomError('This is custom error')
```

Sortie:

```
Traceback (most recent call last):
  File "error_custom.py", line 8, in <module>
    raise CustomError('This is custom error')
__main__.CustomError: This is custom error
```

Attraper une exception personnalisée

Cet exemple montre comment intercepter une exception personnalisée

```
class CustomError(Exception):
     pass

try:
    raise CustomError('Can you catch me ?')
except CustomError as e:
```

```
    print ('Catched CustomError :{}'.format(e))
except Exception as e:
    print ('Generic exception: {}'.format(e))
```

Sortie:

```
Catched CustomError :Can you catch me ?
```

Lire Augmenter les erreurs / exceptions personnalisées en ligne:
https://riptutorial.com/fr/python/topic/10882/augmenter-les-erreurs---exceptions-personnalisees

Chapitre 14: Ballon

Introduction

Flask est un framework Web Python utilisé pour exécuter les principaux sites Web, notamment Pintrest, Twilio et Linkedin. Cette rubrique explique et illustre la diversité des fonctionnalités proposées par Flask pour le développement Web frontal et final.

Syntaxe

- @ app.route ("/ urlpath", methodes = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])
- @ app.route ("/ urlpath / <param>", methodes = ["GET", "POST", "DELETE", "PUTS", "HEAD", "OPTIONS"])

Examples

Les bases

L'exemple suivant est un exemple de serveur de base:

```
# Imports the Flask class
from flask import Flask
# Creates an app and checks if its the main or imported
app = Flask(__name__)

# Specifies what URL triggers hello_world()
@app.route('/')
# The function run on the index route
def hello_world():
    # Returns the text to be displayed
    return "Hello World!"

# If this script isn't an import
if __name__ == "__main__":
    # Run the app until stopped
    app.run()
```

L'exécution de ce script (avec toutes les dépendances correctes installées) devrait démarrer un serveur local. L'hôte est `127.0.0.1` communément appelé **localhost** . Ce serveur s'exécute par défaut sur le port **5000** . Pour accéder à votre serveur Web, ouvrez un navigateur Web et entrez l'URL `localhost:5000` ou `127.0.0.1:5000` (aucune différence). Actuellement, seul votre ordinateur peut accéder au serveur Web.

`app.run()` a trois paramètres, **host** , **port** et **debug** . L'hôte est par défaut `127.0.0.1` , mais si vous le définissez sur `0.0.0.0` , votre serveur Web sera accessible à partir de n'importe quel périphérique de votre réseau en utilisant votre adresse IP privée dans l'URL. le port est par défaut 5000 mais si le paramètre est défini sur le port `80` , les utilisateurs n'auront pas besoin de spécifier

un numéro de port, car les navigateurs utilisent le port 80 par défaut. Comme pour l'option de débogage, pendant le processus de développement (jamais en production), il est utile de définir ce paramètre sur True, car votre serveur redémarrera lorsque des modifications seront apportées à votre projet Flask.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

URL de routage

Avec Flask, le routage d'URL est traditionnellement effectué à l'aide de décorateurs. Ces décorateurs peuvent être utilisés pour le routage statique, ainsi que pour router les URL avec des paramètres. Pour l'exemple suivant, imaginez que ce script Flask exécute le site Web `www.example.com` .

```
@app.route("/")
def index():
    return "You went to www.example.com"

@app.route("/about")
def about():
    return "You went to www.example.com/about"

@app.route("/users/guido-van-rossum")
    return "You went to www.example.com/guido-van-rossum"
```

Avec ce dernier itinéraire, vous pouvez voir que, avec une URL avec / users / et le nom du profil, nous pourrions retourner un profil. Comme il serait horriblement inefficace et désordonné d'inclure un `@app.route()` pour chaque utilisateur, Flask propose de prendre les paramètres depuis l'URL:

```
@app.route("/users/<username>")
def profile(username):
    return "Welcome to the profile of " + username

cities = ["OMAHA", "MELBOURNE", "NEPAL", "STUTTGART", "LIMA", "CAIRO", "SHANGHAI"]

@app.route("/stores/locations/<city>")
def storefronts(city):
    if city in cities:
        return "Yes! We are located in " + city
    else:
        return "No. We are not located in " + city
```

Méthodes HTTP

Les deux méthodes HTTP les plus courantes sont **GET** et **POST** . Flask peut exécuter un code différent à partir de la même URL en fonction de la méthode HTTP utilisée. Par exemple, dans un service Web avec des comptes, il est plus pratique d'acheminer la page de connexion et le processus de connexion via la même URL. Une requête GET, identique à celle qui est effectuée lorsque vous ouvrez une URL dans votre navigateur, doit afficher le formulaire de connexion, tandis qu'une requête POST (contenant des données de connexion) doit être traitée séparément.

Un itinéraire est également créé pour gérer les méthodes DELETE et PUT HTTP.

```
@app.route("/login", methods=["GET"])
def login_form():
    return "This is the login form"
@app.route("/login", methods=["POST"])
def login_auth():
    return "Processing your data"
@app.route("/login", methods=["DELETE", "PUT"])
def deny():
    return "This method is not allowed"
```

Pour simplifier un peu le code, nous pouvons importer le paquet de `request` de flask.

```
from flask import request

@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Processing your data"
```

Pour récupérer des données de la demande POST, nous devons utiliser le package de `request` :

```
from flask import request
@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Username was " + request.form["username"] + " and password was " +
request.form["password"]
```

Fichiers et modèles

Au lieu de taper notre balise HTML dans les instructions de retour, nous pouvons utiliser la fonction `render_template()` :

```
from flask import Flask
from flask import render_template
app = Flask(__name__)

@app.route("/about")
def about():
    return render_template("about-us.html")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

Cela utilisera notre fichier de gabarit `about-us.html` . Pour que notre application puisse trouver ce

fichier, nous devons organiser notre répertoire dans le format suivant:

```
- application.py
/templates
    - about-us.html
    - login-form.html
/static
    /styles
        - about-style.css
        - login-style.css
    /scripts
        - about-script.js
        - login-script.js
```

Plus important encore, les références à ces fichiers dans le code HTML doivent ressembler à ceci:

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-
style.css')}}">
```

qui dirigera l'application à rechercher `about-style.css` dans le dossier styles sous le dossier static. Le même format de chemin s'applique à toutes les références aux images, aux styles, aux scripts ou aux fichiers.

Jinja Templating

Semblable à Meteor.js, Flask s'intègre bien avec les services de gabarit frontaux. Flask utilise par défaut Jinja Templating. Les modèles permettent d'utiliser de petits extraits de code dans le fichier HTML, tels que les conditionnels ou les boucles.

Lorsque nous rendons un modèle, tous les paramètres au-delà du nom du fichier de modèle sont transmis au service de modélisation HTML. L'itinéraire suivant transmettra le nom d'utilisateur et la date jointe (d'une fonction ailleurs) dans le code HTML.

```
@app.route("/users/<username>)
def profile(username):
    joinedDate = get_joined_date(username) # This function's code is irrelevant
    awards = get_awards(username) # This function's code is irrelevant
    # The joinDate is a string and awards is an array of strings
    return render_template("profile.html", username=username, joinDate=joinDate,
awards=awards)
```

Lorsque ce modèle est rendu, il peut utiliser les variables qui lui sont transmises depuis la fonction `render_template()` . Voici le contenu de `profile.html` :

```
<!DOCTYPE html>
<html>
    <head>
        # if username
            <title>Profile of {{ username }}</title>
        # else
            <title>No User Found</title>
        # endif
    <head>
    <body>
```

```
        {% if username %}
            <h1>{{ username }} joined on the date {{ date }}</h1>
            {% if len(awards) > 0 %}
                <h3>{{ username }} has the following awards:</h3>
                <ul>
                {% for award in awards %}
                    <li>{{award}}</li>
                {% endfor %}
                </ul>
            {% else %}
                <h3>{{ username }} has no awards</h3>
            {% endif %}
        {% else %}
            <h1>No user was found under that username</h1>
        {% endif %}
        {# This is a comment and doesn't affect the output #}
    </body>
</html>
```

Les délimiteurs suivants sont utilisés pour différentes interprétations:

- `{% ... %}` indique une déclaration
- `{{ ... }}` désigne une expression où un modèle est généré
- `{# ... #}` indique un commentaire (non inclus dans la sortie du modèle)
- `{# ... ##` implique que le reste de la ligne doit être interprété comme une déclaration

L'objet Request

L'objet `request` fournit des informations sur la requête qui a été effectuée sur l'itinéraire. Pour utiliser cet objet, il faut l'importer depuis le module flask:

```
from flask import request
```

Paramètres d'URL

Dans les exemples précédents, `request.method` et `request.form` ont été utilisés, mais nous pouvons également utiliser la propriété `request.args` pour récupérer un dictionnaire des clés / valeurs dans les paramètres de l'URL.

```
@app.route("/api/users/<username>")
def user_api(username):
    try:
        token = request.args.get("key")
        if key == "pA55w0Rd":
            if isUser(username): # The code of this method is irrelevant
                joined = joinDate(username) # The code of this method is irrelevant
                return "User " + username + " joined on " + joined
            else:
                return "User not found"
        else:
            return "Incorrect key"
    # If there is no key parameter
    except KeyError:
```

```
        return "No key provided"
```

Pour s'authentifier correctement dans ce contexte, l'URL suivante serait nécessaire (en remplaçant le nom d'utilisateur par un nom d'utilisateur quelconque:

`www.example.com/api/users/guido-van-rossum?key=pa55w0Rd`

Téléchargement de fichier

Si un téléchargement de fichier faisait partie du formulaire soumis dans une requête POST, les fichiers peuvent être gérés à l'aide de l'objet `request` :

```
@app.route("/upload", methods=["POST"])
def upload_file():
    f = request.files["wordlist-upload"]
    f.save("/var/www/uploads/" + f.filename) # Store with the original filename
```

Biscuits

La demande peut également inclure des cookies dans un dictionnaire similaire aux paramètres URL.

```
@app.route("/home")
def home():
    try:
        username = request.cookies.get("username")
        return "Your stored username is " + username
    except KeyError:
        return "No username cookies was found")
```

Lire Ballon en ligne: https://riptutorial.com/fr/python/topic/8682/ballon

Chapitre 15: Bibliothèque de sous-processus

Syntaxe

- subprocess.call (args, *, stdin = Aucun, stdout = Aucun, stderr = Aucun, shell = False, timeout = Aucun)
- subprocess.Popen (args, bufsize = -1, exécutable = Aucun, stdin = Aucun, stdout = Aucun, stderr = Aucun, preexec_fn = Aucun, close_fds = True, shell = False, cwd = Aucun, env = Aucun, universal_newlines = False , startupinfo = Aucun, creationflags = 0, restore_signals = True, start_new_session = False, pass_fds = ())

Paramètres

Paramètre	Détails
`args`	Un seul exécutable ou une séquence d'exécutable et d'arguments - `'ls'` , `['ls', '-la']`
`shell`	Courir sous un coquillage? Le shell par défaut vers `/bin/sh` sur POSIX.
`cwd`	Répertoire de travail du processus enfant.

Examples

Appeler des commandes externes

Le cas d'utilisation le plus simple consiste à utiliser la fonction `subprocess.call` . Il accepte une liste comme premier argument. Le premier élément de la liste doit être l'application externe que vous souhaitez appeler. Les autres éléments de la liste sont des arguments qui seront transmis à cette application.

```
subprocess.call([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

Pour les commandes shell, définissez `shell=True` et fournissez la commande sous forme de chaîne au lieu d'une liste.

```
subprocess.call('echo "Hello, world"', shell=True)
```

Notez que les deux commandes ci-dessus ne renvoient que le `exit status` de `exit status` du sous-processus. De plus, faites attention lorsque vous utilisez `shell=True` car il fournit des problèmes de sécurité (voir ici).

Si vous voulez pouvoir obtenir la sortie standard du sous-processus, remplacez le `subprocess.call` par `subprocess.check_output` . Pour une utilisation plus avancée, reportez-vous à ceci .

Plus de souplesse avec Popen

Utiliser `subprocess.Popen` donne un contrôle plus fin sur les processus lancés que `subprocess.call` .

Lancer un sous-processus

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

La signature pour `Popen` est très similaire à la fonction d' `call` ; Cependant, `Popen` reviendra immédiatement au lieu d'attendre que le sous-processus se termine comme l' `call` fait.

En attente d'un sous-processus pour terminer

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
process.wait()
```

Lecture de la sortie d'un sous-processus

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)

# This will block until process completes
stdout, stderr = process.communicate()
print stdout
print stderr
```

Accès interactif aux sous-processus en cours d'exécution

Vous pouvez lire et écrire sur `stdin` et `stdout` même si le sous-processus n'est pas terminé. Cela pourrait être utile lors de l'automatisation des fonctionnalités dans un autre programme.

Ecrire dans un sous-processus

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout = subprocess.PIPE, stdin =
subprocess.PIPE)


process.stdin.write('line of input\n') # Write input
```

```
line  = process.stdout.readline() # Read a line from stdout

# Do logic on line read.
```

Cependant, si vous n'avez besoin que d'un seul ensemble d'entrées et de sorties, plutôt que d'une interaction dynamique, vous devez utiliser `communicate()` plutôt que d'accéder directement à `stdin` et `stdout` .

Lecture d'un flux d'un sous-processus

Si vous souhaitez voir la sortie d'un sous-processus ligne par ligne, vous pouvez utiliser l'extrait suivant:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

Dans le cas où la sortie de la sous-commande n'a pas de caractère EOL, l'extrait ci-dessus ne fonctionne pas. Vous pouvez ensuite lire le caractère de sortie par caractère comme suit:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

Le `1` spécifié comme argument à la `read` méthode demande de lecture pour lire une nature à la fois. Vous pouvez spécifier de lire autant de caractères que vous voulez en utilisant un numéro différent. Le nombre négatif ou 0 indique de `read` pour lire en tant que chaîne unique jusqu'à ce que l'EOF soit rencontré (voir ici).

Dans les deux extraits ci-dessus, `process.poll()` est `None` jusqu'à la fin du sous-processus. Ceci est utilisé pour quitter la boucle une fois qu'il n'y a plus de sortie à lire.

La même procédure pourrait être appliquée au `stderr` du sous-processus.

Comment créer l'argument de la liste de commandes

La méthode de sous-processus qui permet d'exécuter des commandes nécessite la commande sous la forme d'une liste (au moins en utilisant `shell_mode=True`).

Les règles pour créer la liste ne sont pas toujours simples à suivre, en particulier avec les commandes complexes. Heureusement, il existe un outil très utile qui permet de le faire: `shlex` . La manière la plus simple de créer la liste à utiliser comme commande est la suivante:

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

Un exemple simple:

```
import shlex
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

Lire Bibliothèque de sous-processus en ligne:
https://riptutorial.com/fr/python/topic/1393/bibliotheque-de-sous-processus

Chapitre 16: Blocs de code, cadres d'exécution et espaces de noms

Introduction

Un bloc de code est un morceau de texte de programme Python pouvant être exécuté en tant qu'unité, tel qu'un module, une définition de classe ou un corps de fonction. Certains blocs de code (comme les modules) ne sont normalement exécutés qu'une seule fois, d'autres (comme les corps de fonctions) peuvent être exécutés plusieurs fois. Les blocs de code peuvent contenir textuellement d'autres blocs de code. Les blocs de code peuvent invoquer d'autres blocs de code (qui peuvent ou non être contenus textuellement) dans le cadre de leur exécution, par exemple en invoquant (appelant) une fonction.

Examples

Espaces de noms de blocs de code

Type de bloc de code	Espace de noms global	Espace de noms local
Module	ns pour le module	même chose que globale
Script (fichier ou commande)	ns pour `__main__`	même chose que globale
Commande interactive	ns pour `__main__`	même chose que globale
Définition de classe	ns global du bloc contenant	nouvel espace de noms
Corps de fonction	ns global du bloc contenant	nouvel espace de noms
Chaîne transmise à l'instruction `exec`	ns global du bloc contenant	espace de noms local du bloc contenant
Chaîne passée à `eval()`	ns global de l'appelant	ns local de l'appelant
Fichier lu par `execfile()`	ns global de l'appelant	ns local de l'appelant
Expression lue par `input()`	ns global de l'appelant	ns local de l'appelant

Lire Blocs de code, cadres d'exécution et espaces de noms en ligne: https://riptutorial.com/fr/python/topic/10741/blocs-de-code--cadres-d-execution-et-espaces-de-noms

Chapitre 17: Boucles

Introduction

En tant que l'une des fonctions les plus élémentaires de la programmation, les boucles constituent une partie importante de presque tous les langages de programmation. Les boucles permettent aux développeurs de définir certaines parties de leur code à répéter à travers un certain nombre de boucles appelées itérations. Cette rubrique traite de l'utilisation de plusieurs types de boucles et d'applications de boucles dans Python.

Syntaxe

- while <expression booléenne>:
- pour <variable> dans <iterable>:
- pour <variable> dans la plage (<nombre>):
- pour <variable> dans la plage (<numéro_premier>, <numéro_final>):
- pour <variable> dans la plage (<numéro_premier>, <numéro_endmin>, <taille_touche>):
- pour i, <variable> en enumerate (<iterable>): # avec index i
- pour <variable1>, <variable2> dans zip (<iterable1>, <iterable2>):

Paramètres

Paramètre	Détails
Expression booléenne	expression qui peut être évaluée dans un contexte booléen, par exemple `x < 10`
variable	nom de la variable pour l'élément actuel de l' `iterable`
itérable	tout ce qui implémente les itérations

Examples

Itération sur les listes

Pour parcourir une liste, vous pouvez utiliser `for` :

```
for x in ['one', 'two', 'three', 'four']:
    print(x)
```

Cela imprimera les éléments de la liste:

```
one
two
```

```
three
four
```

La fonction `range` génère des nombres qui sont également souvent utilisés dans une boucle for.

```
for x in range(1, 6):
    print(x)
```

Le résultat sera un type de séquence de plage spécial dans python> = 3 et une liste dans python <= 2. Les deux peuvent être bouclés en utilisant la boucle for.

```
1
2
3
4
5
```

Si vous voulez effectuer une boucle à la fois sur les éléments d'une liste *et* avoir un index pour les éléments, vous pouvez utiliser la fonction d' `enumerate` de Python:

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, '::', item)
```

`enumerate` va générer des tuples, décompressés en `index` (entier) et en `item` (valeur réelle de la liste). La boucle ci-dessus s'imprimera

```
(0, '::', 'one')
(1, '::', 'two')
(2, '::', 'three')
(3, '::', 'four')
```

Itérer sur une liste avec manipulation de valeur en utilisant `map` et `lambda` , c'est-à-dire appliquer la fonction lambda sur chaque élément de la liste:

```
x = map(lambda e :  e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

Sortie:

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

NB: dans Python 3.x, la `map` retourne un itérateur au lieu d'une liste, donc vous devez, si vous avez besoin d'une liste, lancer l' `print(list(x))` du résultat `print(list(x))` (voir http://www.riptutorial.com/python/ exemple / 8186 / map-- dans http://www.riptutorial.com/python/topic/809/incompatibilities-moving-from-python-2-to-python-3).

Pour les boucles

`for` boucles, parcourez une collection d'éléments, tels que `list` ou `dict` , et exécutez un bloc de

code avec chaque élément de la collection.

```
for i in [0, 1, 2, 3, 4]:
    print(i)
```

Le ci-dessus `for` boucle itère sur une liste de nombres.

Chaque itération définit la valeur de `i` sur l'élément suivant de la liste. Donc, d'abord, il sera `0` , puis `1` , puis `2` , etc. La sortie sera la suivante:

```
0
1
2
3
4
```

`range` est une fonction qui renvoie une série de nombres sous une forme itérable, ainsi elle peut être utilisée `for` boucles:

```
for i in range(5):
    print(i)
```

donne exactement le même résultat que le premier `for` boucle. Notez que `5` n'est pas imprimé car la plage représente les cinq premiers chiffres à partir de `0` .

Objets à parcourir et itérateurs

`for` loop peut itérer sur tout objet itérable qui est un objet qui définit une fonction `__getitem__` ou `__iter__` . La fonction `__iter__` renvoie un itérateur, qui est un objet avec une fonction `next` utilisée pour accéder à l'élément suivant de l'itérable.

Pause et continuer dans les boucles

déclaration de `break`

Lorsqu'une instruction `break` s'exécute dans une boucle, le flux de contrôle "se casse" immédiatement de la boucle:

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

La condition conditionnelle ne sera pas évaluée après l'exécution de l'instruction `break` . Notez que

les instructions `break` ne sont autorisées qu'à l' *intérieur des boucles* , syntaxiquement. Une instruction `break` dans une fonction ne peut pas être utilisée pour mettre fin aux boucles appelées cette fonction.

L'exécution de la commande suivante imprime chaque chiffre jusqu'au numéro `4` lorsque l'instruction de `break` est remplie et que la boucle s'arrête:

```
0
1
2
3
4
Breaking from loop
```

`break` instructions `break` peuvent également être utilisées à l'intérieur `for` boucles, l'autre construction en boucle fournie par Python:

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

L'exécution de cette boucle imprime maintenant:

```
0
1
2
```

Notez que 3 et 4 ne sont pas imprimés depuis la fin de la boucle.

Si une boucle a une clause `else` , elle ne s'exécute pas lorsque la boucle est terminée par une instruction `break` .

`continue` déclaration

Une instruction `continue` passera à la prochaine itération de la boucle en contournant le reste du bloc en cours mais en continuant la boucle. Comme avec `break` , `continue` ne peut apparaître qu'à l'intérieur des boucles:

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)

0
1
3
5
```

Notez que `2` et `4` ne sont pas imprimés, c'est parce que `continue` va à l'itération suivante au lieu de

continuer à `print(i)` quand `i == 2` ou `i == 4` .

Boucles imbriquées

`break` and `continue` ne fonctionne que sur un seul niveau de boucle. L'exemple suivant ne sortir de l'intérieur `for` la boucle, et non l'extérieur `while` boucle:

```
while True:
    for i in range(1,5):
        if i == 2:
            break    # Will only break out of the inner loop!
```

Python n'a pas la capacité de sortir de plusieurs niveaux de boucle à la fois - si ce comportement est souhaité, le remaniement d'une ou de plusieurs boucles dans une fonction et le remplacement de la `break` avec le `return` peuvent être la solution.

Utiliser le `return` d'une fonction comme une `break`

L' instruction `return` quitte une fonction sans exécuter le code qui suit.

Si vous avez une boucle dans une fonction, utiliser `return` depuis l'intérieur de cette boucle équivaut à avoir une `break` car le reste du code de la boucle n'est pas exécuté (*notez que tout code après la boucle n'est pas exécuté non plus*):

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

Si vous avez des boucles imbriquées, l'instruction `return` interrompt toutes les boucles:

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

va sortir:

```
1 # 1*1
2 # 1*2
3 # 1*3
4 # 1*4
2 # 2*1
4 # 2*2
# return because 2*3 = 6, the remaining iterations of both loops are not executed
```

Boucles avec une clause "else"

Les instructions composites `for` `while`) peuvent avoir une clause `else` (en pratique, cette utilisation est assez rare).

L' `else` clause exécute seulement après une `for` la boucle se termine par itérer jusqu'à la fin, ou après un `while` boucle se termine par son expression conditionnelle devient fausse.

```
for i in range(3):
    print(i)
else:
    print('done')

i = 0
while i < 3:
    print(i)
    i += 1
else:
    print('done')
```

sortie:

```
 0
 1
 2
 done
```

La clause `else` ne s'exécute *pas* si la boucle se termine d'une autre manière (via une instruction `break` ou en levant une exception):

```
for i in range(2):
    print(i)
    if i == 1:
        break
else:
    print('done')
```

sortie:

```
0
1
```

La plupart des autres langages de programmation ne disposent pas de cette clause facultative `else` de boucles. L'utilisation du mot clé `else` en particulier est souvent considérée comme source de confusion.

Le concept original d'une telle clause remonte à Donald Knuth et la signification du mot clé `else` devient claire si nous réécrivons une boucle en termes d'instructions `if` et `goto` des jours précédents avant la programmation structurée ou à partir d'un langage d'assemblage de niveau inférieur.

Par exemple:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
```

est équivalent à:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>

<<end>>:
```

Celles-ci restent équivalentes si nous attachons une clause `else` à chacune d'elles.

Par exemple:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```

est équivalent à:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

Une boucle `for` avec une clause `else` peut être comprise de la même façon. D'un point de vue conceptuel, il existe une condition de boucle qui reste vraie tant que l'objet ou la séquence itérable contient encore quelques éléments.

Pourquoi utiliserait cette construction étrange?

Le principal cas d'utilisation de la construction `for...else` est une implémentation concise de la recherche, par exemple:

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

Pour rendre le `else` dans cette construction moins déroutant, on peut le considérer comme " *sinon casser* " ou " *si non trouvé* ".

Quelques discussions à ce sujet peuvent être trouvées dans [Python-ideas] Résumé de pour ... Threads , Pourquoi python utilise-t-il les boucles after pour et while? et autres clauses sur les instructions de boucle

Itération sur les dictionnaires

Considérant le dictionnaire suivant:

```
 d = {"a": 1, "b": 2, "c": 3}
```

Pour parcourir ses clés, vous pouvez utiliser:

```
for key in d:
    print(key)
```

Sortie:

```
"a"
"b"
"c"
```

Ceci est équivalent à:

```
for key in d.keys():
    print(key)
```

ou en Python 2:

```
for key in d.iterkeys():
    print(key)
```

Pour parcourir ses valeurs, utilisez:

```
for value in d.values():
```

```
    print(value)
```

Sortie:

```
1
2
3
```

Pour parcourir ses clés et ses valeurs, utilisez:

```
for key, value in d.items():
    print(key, "::", value)
```

Sortie:

```
a :: 1
b :: 2
c :: 3
```

Notez que dans Python 2, `.keys()` , `.values()` et `.items()` renvoient un objet `list` . Si vous devez simplement parcourir le résultat, vous pouvez utiliser les équivalents `.iterkeys()` , `.itervalues()` et `.iteritems()` .

La différence entre `.keys()` et `.iterkeys()` , `.values()` et `.itervalues()` , `.items()` et `.iteritems()` est que les méthodes `iter*` sont des générateurs. Ainsi, les éléments du dictionnaire sont fournis un par un au fur et à mesure de leur évaluation. Lorsqu'un objet `list` est renvoyé, tous les éléments sont regroupés dans une liste, puis renvoyés pour une évaluation ultérieure.

> Notez également que dans Python 3, l'ordre des éléments imprimés de la manière ci-dessus ne suit aucun ordre.

En boucle

A `while` boucle ramènerai les instructions de boucle à exécuter jusqu'à ce que la condition de la boucle est Falsey . Le code suivant exécutera les instructions de boucle 4 fois au total.

```
i = 0
while i < 4:
    #loop statements
    i = i + 1
```

Alors que la boucle ci-dessus peut facilement être convertie en une boucle `for` élégante, `while` boucles sont utiles pour vérifier si certaines conditions ont été remplies. La boucle suivante continuera à s'exécuter jusqu'à `myObject` que `myObject` soit prêt.

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

`while` boucles `while` peuvent également s'exécuter sans condition en utilisant des nombres (complexes ou réels) ou `True` :

```
import cmath

complex_num = cmath.sqrt(-1)
while complex_num:      # You can also replace complex_num with any number, True or a value of
any type
    print(complex_num)   # Prints 1j forever
```

Si la condition est toujours vraie, la boucle while sera exécutée pour toujours (boucle infinie) si elle n'est pas terminée par une instruction break ou return ou une exception.

```
while True:
    print "Infinite loop"
# Infinite loop
# Infinite loop
# Infinite loop
# ...
```

La déclaration de passage

`pass` est une instruction vide pour lorsqu'une instruction est requise par la syntaxe python (comme dans le corps d'un `for` ou `while` la boucle), mais aucune action est nécessaire ou souhaitée par le programmeur. Cela peut être utile comme espace réservé pour le code qui doit encore être écrit.

```
for x in range(10):
    pass #we don't want to do anything, or are not ready to do anything here, so we'll pass
```

Dans cet exemple, rien ne se passera. La boucle `for` se terminera sans erreur, mais aucune commande ou code ne sera activé. `pass` nous permet d'exécuter notre code avec succès sans que toutes les commandes et actions soient complètement implémentées.

De même, `pass` peut être utilisé dans les boucles `while` , ainsi que dans les sélections et les définitions de fonctions, etc.

```
while x == y:
    pass
```

Itérer différentes parties d'une liste avec différentes tailles de pas

Supposons que vous ayez une longue liste d'éléments et que vous ne soyez intéressé que par tous les autres éléments de la liste. Vous souhaitez peut-être examiner uniquement le premier ou le dernier élément, ou une plage spécifique d'entrées dans votre liste. Python possède de puissantes fonctionnalités intégrées d'indexation. Voici quelques exemples de réalisation de ces scénarios.

Voici une liste simple qui sera utilisée dans tous les exemples:

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

Itération sur toute la liste

Pour parcourir chaque élément de la liste, une boucle `for` comme ci-dessous peut être utilisée:

```
for s in lst:
    print s[:1] # print the first letter
```

La boucle `for` assigne des s à chaque élément de `lst` . Cela va imprimer:

```
a
b
c
d
e
```

Vous avez souvent besoin à la fois de l'élément et de l'index de cet élément. Le mot clé `enumerate` exécute cette tâche.

```
for idx, s in enumerate(lst):
    print("%s has an index of %d" % (s, idx))
```

L'index `idx` commencera par zéro et par incrément pour chaque itération, tandis que le `s` contiendra l'élément en cours de traitement. L'extrait de code précédent affichera:

```
alpha has an index of 0
bravo has an index of 1
charlie has an index of 2
delta has an index of 3
echo has an index of 4
```

Itérer sur la sous-liste

Si vous souhaitez effectuer une itération sur une plage (en vous rappelant que Python utilise l'indexation basée sur zéro), utilisez le mot clé `range` .

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

Cela produirait:

```
lst at 2 contains charlie
lst at 3 contains delta
```

La liste peut également être tranchée. La notation de tranche suivante va d'un élément à l'index 1

à la fin avec un pas de 2. Les deux `for` boucles donnent le même résultat.

```
for s in lst[1::2]:
    print(s)

for i in range(1, len(lst), 2):
    print(lst[i])
```

Les extraits de code ci-dessus:

```
bravo
delta
```

L'indexation et le découpage sont un sujet à part.

Le "demi-boucle" à faire

Contrairement à d'autres langages, Python n'a pas de construction à faire ou à faire (cela permettra l'exécution du code une fois avant le test de la condition). Cependant, vous pouvez combiner un `while True` avec un `break` pour atteindre le même objectif.

```
a = 10
while True:
    a = a-1
    print(a)
    if a<7:
        break
print('Done.')
```

Cela va imprimer:

```
9
8
7
6
Done.
```

En boucle et déballage

Si vous souhaitez parcourir une liste de tuples par exemple:

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

au lieu de faire quelque chose comme ça:

```
for item in collection:
    i1 = item[0]
    i2 = item[1]
    i3 = item[2]
    # logic
```

ou quelque chose comme ça:

```
for item in collection:
    i1, i2, i3 = item
    # logic
```

Vous pouvez simplement faire ceci:

```
for i1, i2, i3 in collection:
    # logic
```

Cela fonctionnera également pour la *plupart des* types d'itérables, pas seulement les tuples.

Lire Boucles en ligne: https://riptutorial.com/fr/python/topic/237/boucles

Chapitre 18: Calcul parallèle

Remarques

En raison du GIL (Global Interpreter Lock), une seule instance de l'interpréteur python s'exécute en un seul processus. Donc, en général, l'utilisation du multi-threading n'améliore que les calculs liés aux entrées-sorties, pas ceux liés au processeur. Le module de `multiprocessing` est recommandé si vous souhaitez paralléliser les tâches liées au processeur.

GIL s'applique à CPython, l'implémentation la plus populaire de Python, ainsi que PyPy. D'autres implémentations telles que Jython et IronPython n'ont pas de GIL .

Examples

Utilisation du module de multitraitement pour paralléliser des tâches

```
import multiprocessing

def fib(n):
    """computing the Fibonacci in an inefficient way
    was chosen to slow down the CPU."""
    if n <= 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
p = multiprocessing.Pool()
print(p.map(fib,[38,37,36,35,34,33]))

# Out: [39088169, 24157817, 14930352, 9227465, 5702887, 3524578]
```

Comme l'exécution de chaque appel à `fib` se déroule en parallèle, le temps d'exécution de l'exemple complet est **1,8 fois plus rapide** que s'il était exécuté de manière séquentielle sur un processeur double.

Python 2.2+

Utiliser les scripts Parent et Children pour exécuter du code en parallèle

child.py

```
import time

def main():
    print "starting work"
    time.sleep(1)
    print "work work work work work"
    time.sleep(1)
    print "done working"

if __name__ == '__main__':
```

```
    main()
```

parent.py

```
import os

def main():
    for i in range(5):
        os.system("python child.py &")

if __name__ == '__main__':
    main()
```

Ceci est utile pour les tâches de requête / réponse HTTP parallèles et indépendantes ou pour les sélections / insertions de base de données. Des arguments de ligne de commande peuvent également être donnés au script **child.py** . La synchronisation entre les scripts peut être réalisée par tous les scripts vérifiant régulièrement un serveur distinct (comme une instance Redis).

Utiliser une extension C pour paralléliser des tâches

L'idée ici est de déplacer les travaux intensifs en calcul vers C (en utilisant des macros spéciales), indépendamment de Python, et de faire en sorte que le code C libère le GIL pendant qu'il fonctionne.

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

Utiliser le module PyPar pour paralléliser

PyPar est une bibliothèque qui utilise l'interface de transmission de messages (MPI) pour fournir le parallélisme en Python. Un exemple simple dans PyPar (vu à https://github.com/daleroberts/pypar) ressemble à ceci:

```
import pypar as pp

ncpus = pp.size()
rank = pp.rank()
node = pp.get_processor_name()

print 'I am rank %d of %d on node %s' % (rank, ncpus, node)

if rank == 0:
  msh = 'P0'
  pp.send(msg, destination=1)
  msg = pp.receive(source=rank-1)
```

```
  print 'Processor 0 received message "%s" from rank %d' % (msg, rank-1)
else:
  source = rank-1
  destination = (rank+1) % ncpus
  msg = pp.receive(source)
  msg = msg + 'P' + str(rank)
  pypar.send(msg, destination)
pp.finalize()
```

Lire Calcul parallèle en ligne: https://riptutorial.com/fr/python/topic/542/calcul-parallele

Chapitre 19: Caractéristiques cachées

Examples

Surcharge de l'opérateur

Tout en Python est un objet. Chaque objet possède des méthodes internes spéciales qu'il utilise pour interagir avec d'autres objets. Généralement, ces méthodes suivent la convention de nommage `__action__` . Collectivement, cela s'appelle le modèle de données Python .

Vous pouvez surcharger l' *une* de ces méthodes. Ceci est couramment utilisé dans la surcharge des opérateurs en Python. Vous trouverez ci-dessous un exemple de surcharge d'opérateur utilisant le modèle de données Python. La classe `Vector` crée un vecteur simple de deux variables. Nous allons ajouter un support approprié pour les opérations mathématiques de deux vecteurs en utilisant la surcharge de l'opérateur.

```
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, v):
        # Addition with another vector.
        return Vector(self.x + v.x, self.y + v.y)

    def __sub__(self, v):
        # Subtraction with another vector.
        return Vector(self.x - v.x, self.y - v.y)

    def __mul__(self, s):
        # Multiplication with a scalar.
        return Vector(self.x * s, self.y * s)

    def __div__(self, s):
        # Division with a scalar.
        float_s = float(s)
        return Vector(self.x / float_s, self.y / float_s)

    def __floordiv__(self, s):
        # Division with a scalar (value floored).
        return Vector(self.x // s, self.y // s)

    def __repr__(self):
        # Print friendly representation of Vector class. Else, it would
        # show up like, <__main__.Vector instance at 0x01DDDDC8>.
        return '<Vector (%f, %f)>' % (self.x, self.y, )

a = Vector(3, 5)
b = Vector(2, 7)

print a + b # Output: <Vector (5.000000, 12.000000)>
print b - a # Output: <Vector (-1.000000, 2.000000)>
print b * 1.3 # Output: <Vector (2.600000, 9.100000)>
print a // 17 # Output: <Vector (0.000000, 0.000000)>
```

```
print a / 17 # Output: <Vector (0.176471, 0.294118)>
```

L'exemple ci-dessus montre une surcharge des opérateurs numériques de base. Une liste complète peut être trouvée ici .

Lire Caractéristiques cachées en ligne: https://riptutorial.com/fr/python/topic/946/caracteristiques-cachees

Chapitre 20: ChemPy - package python

Introduction

ChemPy est un package Python conçu principalement pour résoudre et résoudre des problèmes de chimie physique, analytique et inorganique. C'est une boîte à outils Python gratuite et open-source pour les applications de chimie, de génie chimique et de science des matériaux.

Examples

Formules d'analyse

```
from chempy import Substance
ferricyanide = Substance.from_formula('Fe(CN)6-3')
ferricyanide.composition == {0: -3, 26: 1, 6: 6, 7: 6}
True
print(ferricyanide.unicode_name)
Fe(CN)₆³⁻
 print(ferricyanide.latex_name + ", " + ferricyanide.html_name)
Fe(CN)_{6}^{3-}, Fe(CN)<sub>6</sub><sup>3-</sup>
 print('%.3f' % ferricyanide.mass)
211.955
```

Dans la composition, les numéros atomiques (et 0 pour la charge) sont utilisés comme clés et le nombre de chaque type est devenu valeur respective.

Équilibrer la stoechiométrie d'une réaction chimique

```
 from chempy import balance_stoichiometry  # Main reaction in NASA's booster rockets:
 reac, prod = balance_stoichiometry({'NH4ClO4', 'Al'}, {'Al2O3', 'HCl', 'H2O', 'N2'})
 from pprint import pprint
 pprint(reac)
{'Al': 10, 'NH4ClO4': 6}
 pprint(prod)
{'Al2O3': 5, 'H2O': 9, 'HCl': 6, 'N2': 3}
 from chempy import mass_fractions
 for fractions in map(mass_fractions, [reac, prod]):
...     pprint({k: '{0:.3g} wt%'.format(v*100) for k, v in fractions.items()})
...
{'Al': '27.7 wt%', 'NH4ClO4': '72.3 wt%'}
{'Al2O3': '52.3 wt%', 'H2O': '16.6 wt%', 'HCl': '22.4 wt%', 'N2': '8.62 wt%'}
```

Équilibrer les réactions

```
from chempy import Equilibrium
 from sympy import symbols
 K1, K2, Kw = symbols('K1 K2 Kw')
 e1 = Equilibrium({'MnO4-': 1, 'H+': 8, 'e-': 5}, {'Mn+2': 1, 'H2O': 4}, K1)
 e2 = Equilibrium({'O2': 1, 'H2O': 2, 'e-': 4}, {'OH-': 4}, K2)
 coeff = Equilibrium.eliminate([e1, e2], 'e-')
```

```
 coeff
[4, -5]
 redox = e1*coeff[0] + e2*coeff[1]
 print(redox)
20 OH- + 32 H+ + 4 MnO4- = 26 H2O + 4 Mn+2 + 5 O2; K1**4/K2**5
 autoprot = Equilibrium({'H2O': 1}, {'H+': 1, 'OH-': 1}, Kw)
 n = redox.cancel(autoprot)
 n
20
 redox2 = redox + n*autoprot
 print(redox2)
12 H+ + 4 MnO4- = 4 Mn+2 + 5 O2 + 6 H2O; K1**4*Kw**20/K2**5
```

Équilibres chimiques

```
 from chempy import Equilibrium
 from chempy.chemistry import Species
 water_autop = Equilibrium({'H2O'}, {'H+', 'OH-'}, 10**-14)  # unit "molar" assumed
 ammonia_prot = Equilibrium({'NH4+'}, {'NH3', 'H+'}, 10**-9.24)  # same here
 from chempy.equilibria import EqSystem
 substances = map(Species.from_formula, 'H2O OH- H+ NH3 NH4+'.split())
 eqsys = EqSystem([water_autop, ammonia_prot], substances)
 print('\n'.join(map(str, eqsys.rxns)))  # "rxns" short for "reactions"
H2O = H+ + OH-; 1e-14
NH4+ = H+ + NH3; 5.75e-10
 from collections import defaultdict
 init_conc = defaultdict(float, {'H2O': 1, 'NH3': 0.1})
 x, sol, sane = eqsys.root(init_conc)
 assert sol['success'] and sane
 print(sorted(sol.keys()))  # see package "pyneqsys" for more info
['fun', 'intermediate_info', 'internal_x_vecs', 'nfev', 'njev', 'success', 'x', 'x_vecs']
 print(', '.join('%.2g' % v for v in x))
1, 0.0013, 7.6e-12, 0.099, 0.0013
```

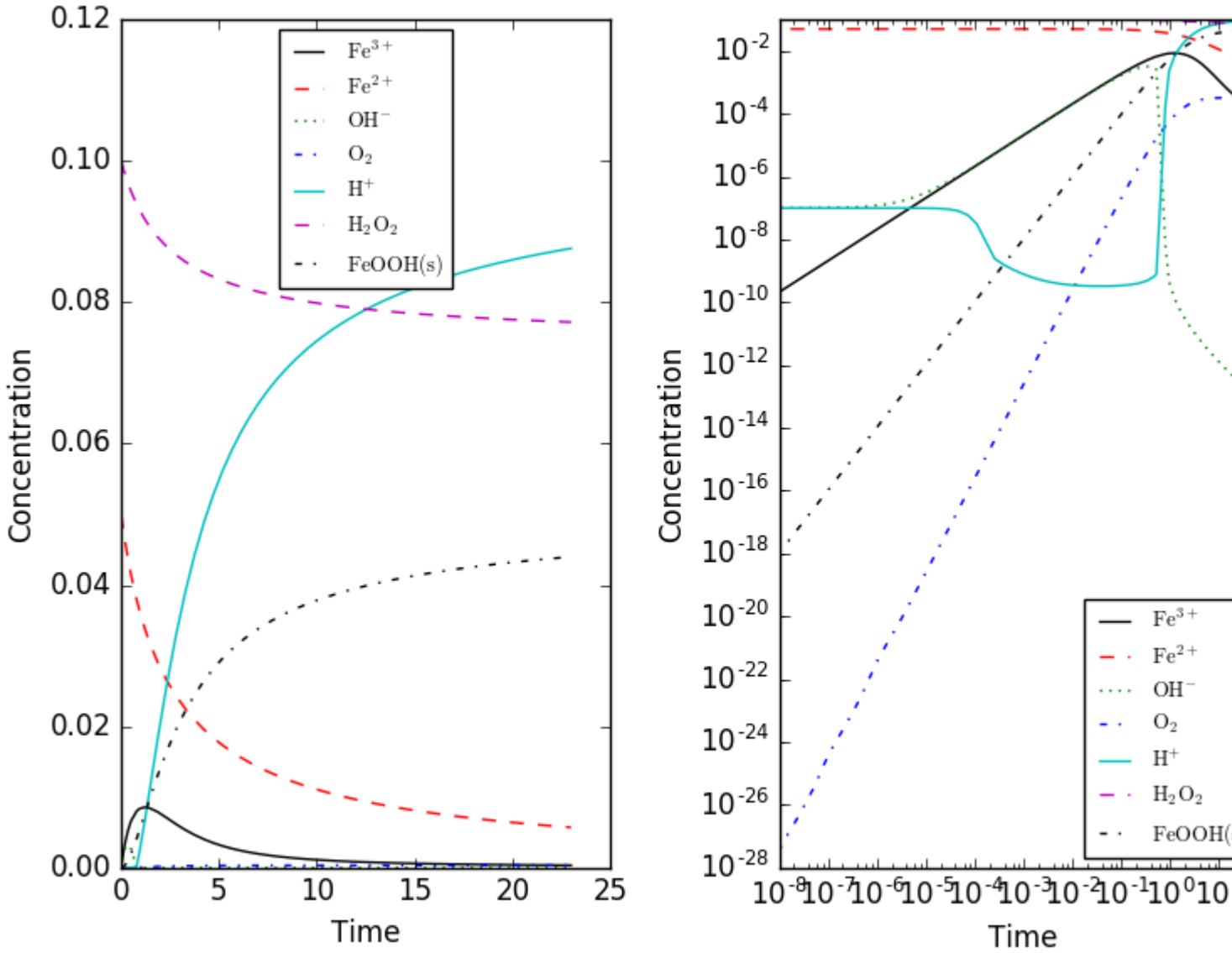
Force ionique

```
 from chempy.electrolytes import ionic_strength
     ionic_strength({'Fe+3': 0.050, 'ClO4-': 0.150}) == .3
    True
```

Cinétique chimique (système d'équations différentielles ordinaires)

```
from chempy import ReactionSystem  # The rate constants below are arbitrary
 rsys = ReactionSystem.from_string("""2 Fe+2 + H2O2 -> 2 Fe+3 + 2 OH-; 42
     2 Fe+3 + H2O2 -> 2 Fe+2 + O2 + 2 H+; 17
     H+ + OH- -> H2O; 1e10
     H2O -> H+ + OH-; 1e-4
     Fe+3 + 2 H2O -> FeOOH(s) + 3 H+; 1
     FeOOH(s) + 3 H+ -> Fe+3 + 2 H2O; 2.5""")  # "[H2O]" = 1.0 (actually 55.4 at RT)
 from chempy.kinetics.ode import get_odesys
 odesys, extra = get_odesys(rsys)
 from collections import defaultdict
 import numpy as np
 tout = sorted(np.concatenate((np.linspace(0, 23), np.logspace(-8, 1))))
 c0 = defaultdict(float, {'Fe+2': 0.05, 'H2O2': 0.1, 'H2O': 1.0, 'H+': 1e-7, 'OH-': 1e-7})
 result = odesys.integrate(tout, c0, atol=1e-12, rtol=1e-14)
```

```
 import matplotlib.pyplot as plt
 _ = plt.subplot(1, 2, 1)
 _ = result.plot(names=[k for k in rsys.substances if k != 'H2O'])
 _ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
 _ = plt.subplot(1, 2, 2)
 _ = result.plot(names=[k for k in rsys.substances if k != 'H2O'], xscale='log', yscale='log')
 _ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
 _ = plt.tight_layout()
 plt.show()
```



Lire ChemPy - package python en ligne: https://riptutorial.com/fr/python/topic/10625/chempy---package-python

Chapitre 21: Classes d'extension et d'extension

Examples

Mixins

Dans un langage de programmation orienté objet, un mixin est une classe qui contient des méthodes à utiliser par d'autres classes sans devoir être la classe parente de ces autres classes. La façon dont ces autres classes accèdent aux méthodes de mixin dépend de la langue.

Il fournit un mécanisme pour l'héritage multiple en permettant à plusieurs classes d'utiliser les fonctionnalités communes, mais sans la sémantique complexe de l'héritage multiple. Les mixins sont utiles lorsqu'un programmeur souhaite partager des fonctionnalités entre différentes classes. Au lieu de répéter le même code encore et encore, les fonctionnalités communes peuvent simplement être regroupées dans un mixin, puis héritées dans chaque classe qui le requiert.

Lorsque l'on utilise plus d'un mixin, l'ordre des mixins est important. Voici un exemple simple:

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(Mixin1, Mixin2):
    pass
```

Dans cet exemple, nous appelons `MyClass` et la méthode de `test` ,

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

Le résultat doit être Mixin1 car Order est de gauche à droite. Cela pourrait donner des résultats inattendus lorsque les super-classes y sont associées. Donc, l'ordre inverse est plus juste comme ceci:

```
class MyClass(Mixin2, Mixin1):
    pass
```

Le résultat sera:

```
>>> obj = MyClass()
>>> obj.test()
```

```
Mixin2
```

Les mixins peuvent être utilisés pour définir des plug-ins personnalisés.

Python 3.x 3.0

```
class Base(object):
    def test(self):
        print("Base.")

class PluginA(object):
    def test(self):
        super().test()
        print("Plugin A.")

class PluginB(object):
    def test(self):
        super().test()
        print("Plugin B.")

plugins = PluginA, PluginB

class PluginSystemA(PluginA, Base):
    pass

class PluginSystemB(PluginB, Base):
    pass

PluginSystemA().test()
# Base.
# Plugin A.

 PluginSystemB().test()
# Base.
# Plugin B.
```

Plugins avec des classes personnalisées

Dans Python 3.6, PEP 487 a ajouté la méthode spéciale `__init_subclass__` , qui simplifie et étend la personnalisation de classe sans utiliser de métaclasses . Par conséquent, cette fonctionnalité permet de créer des plugins simples . Ici, nous démontrons cette fonctionnalité en modifiant un exemple précédent :

Python 3.x 3.6

```
class Base:
    plugins = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.plugins.append(cls)

    def test(self):
        print("Base.")

class PluginA(Base):
    def test(self):
```

```
        super().test()
        print("Plugin A.")


class PluginB(Base):
    def test(self):
        super().test()
        print("Plugin B.")
```

Résultats:

```
PluginA().test()
# Base.
# Plugin A.

PluginB().test()
# Base.
# Plugin B.

Base.plugins
# [__main__.PluginA, __main__.PluginB]
```

Lire Classes d'extension et d'extension en ligne:
https://riptutorial.com/fr/python/topic/4724/classes-d-extension-et-d-extension

Chapitre 22: Classes de base abstraites (abc)

Examples

Définition de la métaclasse ABCMeta

Les classes abstraites sont des classes destinées à être héritées mais évitent d'implémenter des méthodes spécifiques, ne laissant que les signatures de méthode que les sous-classes doivent implémenter.

Les classes abstraites sont utiles pour définir et appliquer des abstractions de classe à un niveau élevé, similaire au concept d'interfaces dans des langages typés, sans qu'il soit nécessaire d'implémenter une méthode.

Une approche conceptuelle pour définir une classe abstraite consiste à extraire les méthodes de classe, puis à générer une erreur NotImplementedError si on y accède. Cela empêche les classes enfants d'accéder aux méthodes parentes sans les remplacer en premier. Ainsi:

```
class Fruit:

    def check_ripeness(self):
        raise NotImplementedError("check_ripeness method not implemented!")


class Apple(Fruit):
    pass


a = Apple()
a.check_ripeness() # raises NotImplementedError
```

La création d'une classe abstraite de cette manière empêche l'utilisation inappropriée de méthodes qui ne sont pas remplacées, et encourage certainement les méthodes à définir dans les classes enfants, mais elle n'applique pas leur définition. Avec le module `abc` nous pouvons empêcher les classes enfants d'être instanciées lorsqu'elles ne remplacent pas les méthodes de classes abstraites de leurs parents et ancêtres:

```
from abc import ABCMeta

class AbstractClass(object):
    # the metaclass attribute must always be set as a class variable
    __metaclass__ = ABCMeta

   # the abstractmethod decorator registers this method as undefined
   @abstractmethod
   def virtual_method_subclasses_must_define(self):
       # Can be left completely blank, or a base implementation can be provided
       # Note that ordinarily a blank interpretation implicitly returns `None`,
       # but by registering, this behaviour is no longer enforced.
```

Il est maintenant possible de simplement sous-classer et remplacer:

```
class Subclass(AbstractClass):
    def virtual_method_subclasses_must_define(self):
        return
```

Pourquoi / Comment utiliser ABCMeta et @abstractmethod

Les classes de base abstraites (ABC) imposent les classes dérivées qui implémentent des méthodes particulières de la classe de base.

Pour comprendre comment cela fonctionne et pourquoi nous devrions l'utiliser, jetons un coup d'œil à un exemple que Van Rossum apprécierait. Disons que nous avons une classe de base "MontyPython" avec deux méthodes (blague & ligne de frappe) qui doivent être implémentées par toutes les classes dérivées.

```
class MontyPython:
    def joke(self):
        raise NotImplementedError()

    def punchline(self):
        raise NotImplementedError()

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

Lorsque nous instancions un objet et appelons deux méthodes, nous obtenons une erreur (comme prévu) avec la méthode `punchline()` .

```
 >>> sketch = ArgumentClinic()
 >>> sketch.punchline()
NotImplementedError
```

Cependant, cela nous permet toujours d'instancier un objet de la classe ArgumentClinic sans obtenir d'erreur. En fait, nous n'obtenons pas d'erreur tant que nous ne cherchons pas la punchline ().

Ceci est évité en utilisant le module ABC (Abstract Base Class). Voyons comment cela fonctionne avec le même exemple:

```
from abc import ABCMeta, abstractmethod

class MontyPython(metaclass=ABCMeta):
    @abstractmethod
    def joke(self):
        pass

@abstractmethod
def punchline(self):
    pass
```

```
class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

Cette fois, lorsque nous essayons d'instancier un objet à partir de la classe incomplète, nous obtenons immédiatement une erreur TypeError!

```
>>> c = ArgumentClinic()
TypeError:
"Can't instantiate abstract class ArgumentClinic with abstract methods punchline"
```

Dans ce cas, il est facile de terminer la classe pour éviter tout type d'erreur:

```
class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"

    def punchline(self):
        return "Send in the constable!"
```

Cette fois, lorsque vous instanciez un objet, cela fonctionne!

Lire Classes de base abstraites (abc) en ligne: https://riptutorial.com/fr/python/topic/5442/classes-de-base-abstraites--abc-

Chapitre 23: Collecte des ordures

Remarques

À la base, le ramasse-miettes de Python (à partir de 3.5) est une simple implémentation de comptage de références. Chaque fois que vous faites une référence à un objet (par exemple, `a = myobject`), le nombre de références sur cet objet (myobject) est incrémenté. Chaque fois qu'une référence est supprimée, le compte de référence est décrémenté, et une fois que le compte de référence atteint `0` , nous savons que rien ne contient de référence à cet objet et que nous pouvons le désallouer!

Un des malentendus courants concernant le fonctionnement de la gestion de la mémoire Python est que le mot-clé `del` libère la mémoire des objets. Ce n'est pas vrai. En fait, le mot-clé `del` ne fait que décrémenter les objets refcount, ce qui signifie que si vous l'appelez suffisamment de fois pour que le refcount atteigne zéro, l'objet peut être récupéré (même s'il existe encore des références à l'objet disponible ailleurs dans votre code)).

Python crée ou nettoie agressivement des objets la première fois qu'il en a besoin Si j'effectue l'affectation a = object (), la mémoire de l'objet est allouée à ce moment (cpython réutilisera parfois certains types d'objet, par exemple des listes sous le capot). mais la plupart du temps, il ne conserve pas de pool d'objets gratuits et effectue des allocations quand vous en avez besoin. De même, dès que le refcount est décrémenté à 0, GC le nettoie.

Collecte de déchets générationnelle

Dans les années 1960, John McCarthy a découvert une erreur fatale dans le refcounting de garbage collection lorsqu'il implémentait l'algorithme de refcounting utilisé par Lisp: que se passe-t-il si deux objets se réfèrent l'un à l'autre dans une référence cyclique? Comment pouvez-vous jamais ramasser ces deux objets même s'il n'y a pas de références externes à ces objets s'ils se réfèrent toujours l'un à l'autre? Ce problème s'étend également à toute structure de données cyclique, telle que les tampons en anneau ou deux entrées consécutives dans une liste à double liaison. Python tente de résoudre ce problème en utilisant une variante légèrement intéressante d'un autre algorithme de récupération de place appelé **Generational Garbage Collection** .

Essentiellement, chaque fois que vous créez un objet en Python, il l'ajoute à la fin d'une liste doublement liée. À l'occasion, Python fait une boucle dans cette liste, vérifie quels objets font référence aux objets de la liste et, s'ils se trouvent également dans la liste (nous verrons pourquoi ils ne le seront pas dans un instant), réduit davantage leurs refcounts. À ce stade (en fait, certaines heuristiques déterminent le moment où les choses sont déplacées, mais supposons que, après une seule collection, les choses restent simples), tout ce qui a un refcount supérieur à 0 est promu dans une autre liste appelée "Génération 1" (C'est pourquoi tous les objets ne figurent pas toujours dans la liste de génération 0), qui a cette boucle appliquée moins souvent. C'est ici qu'intervient la récupération de mémoire générationnelle. Il existe 3 générations par défaut dans Python (trois listes d'objets liées): La première liste (génération 0) contient tous les nouveaux objets; Si un cycle de CPG se produit et que les objets ne sont pas collectés, ils sont

déplacés vers la deuxième liste (génération 1) et si un cycle de CP survient sur la deuxième liste et qu'ils ne sont toujours pas collectés, ils sont déplacés vers la troisième liste (génération 2)). La liste de troisième génération (appelée "génération 2", puisque nous avons une indexation nulle) est beaucoup moins souvent collectée que les deux premières, l'idée étant que si votre objet a une longue vie, il est peu probable être GCed pendant la durée de vie de votre application, il est donc inutile de perdre du temps à la vérifier sur chaque cycle de GC. De plus, on observe que la plupart des objets sont collectés relativement rapidement. A partir de maintenant, nous appellerons ces "bons objets" car ils meurent jeunes. Cela s'appelle "l'hypothèse générationnelle faible" et a également été observé pour la première fois dans les années soixante.

Une mise de côté rapide: contrairement aux deux premières générations, la liste de troisième génération de longue durée de vie n'est pas une collecte régulière des ordures. Il est vérifié lorsque le rapport entre les objets en attente de longue durée (ceux qui figurent dans la liste de troisième génération, mais qui n'ont pas encore eu de cycle GC) avec le nombre total d'objets à vie longue de la liste est supérieur à 25%. C'est parce que la troisième liste est illimitée (les choses ne sont jamais déplacées d'une autre liste, donc elles ne disparaissent que lorsqu'elles sont réellement récupérées), ce qui signifie que pour les applications où vous créez beaucoup d'objets à vie longue sur la troisième liste peut être assez long. En utilisant un ratio, nous obtenons une "performance linéaire amortie dans le nombre total d'objets"; alias, plus la liste est longue, plus le GC prend de temps, mais moins nous effectuons de GC (voici la proposition originale de 2008 pour cette heuristique de Martin von Löwis). Le fait d'effectuer une récupération de place sur la troisième génération ou la liste "mature" est appelée "récupération complète".

La récupération de place générationnelle accélère donc les choses en n'exigeant pas que nous analysions des objets qui n'auront probablement pas besoin de GC tout le temps, mais comment cela nous aide-t-il à casser des références cycliques? Probablement pas très bien, il se trouve que La fonction de rupture de ces cycles de référence commence ainsi :

```
/* Break reference cycles by clearing the containers involved.  This is
 * tricky business as the lists can be changing and we don't know which
 * objects may be freed.  It is possible I screwed something up here.
 */
static void
delete_garbage(PyGC_Head *collectable, PyGC_Head *old)
```

La raison pour laquelle la récupération de la mémoire générationnelle aide à cela est que nous pouvons conserver la longueur de la liste en tant que compte distinct; à chaque fois que nous ajoutons un nouvel objet à la génération, nous incrémentons ce nombre, et chaque fois que nous déplaçons un objet vers une autre génération ou un dealloc, nous décrémentons le nombre. Théoriquement, à la fin d'un cycle de CPG, ce compte (pour les deux premières générations de toute façon) devrait toujours être 0. Si ce n'est pas le cas, tout élément de la liste est une forme de référence circulaire. Cependant, il y a un autre problème ici: que se passe-t-il si les objets restants ont la méthode magique de Python `__del__` sur eux? `__del__` est appelé à chaque fois qu'un objet Python est détruit. Cependant, si deux objets dans une référence circulaire ont des méthodes `__del__` , nous ne pouvons pas être sûrs que leur destruction ne va pas casser la méthode `__del__` autres. Pour un exemple artificiel, imaginons que nous avons écrit ce qui suit:

```
class A(object):
```

```
    def __init__(self, b=None):
        self.b = b

    def __del__(self):
        print("We're deleting an instance of A containing:", self.b)

class B(object):
    def __init__(self, a=None):
        self.a = a

    def __del__(self):
        print("We're deleting an instance of B containing:", self.a)
```

et nous définissons une instance de A et une instance de B pour qu'elles se pointent l'une sur l'autre et qu'elles se retrouvent dans le même cycle de récupération de place? Disons que nous en sélectionnons un au hasard et distribuons notre instance de A en premier; La méthode `__del__` de A sera appelée, elle sera imprimée, puis A sera libérée. Ensuite, nous arrivons à B, nous appelons sa méthode `__del__` , et oops! Segfault! A n'existe plus. Nous pourrions résoudre ce problème en appelant tout ce qui reste des méthodes `__del__` , puis en effectuant une autre passe pour tout désassembler. Cependant, ceci introduit un autre problème: si une méthode `__del__` enregistre une référence de l'autre objet a une référence à nous ailleurs? Nous avons toujours un cycle de référence, mais maintenant il n'est plus possible de GC non plus l'objet, même s'ils ne sont plus utilisés. Notez que même si un objet ne fait pas partie d'une structure de données circulaire, il pourrait se `__del__` dans sa propre méthode `__del__` ; Python vérifie cela et arrêtera GCing si un refcount d'objets a augmenté après l' `__del__` sa méthode `__del__` .

CPython s'occupe de cela en collant ces objets non-GC-possibles (n'importe quoi avec une forme de référence circulaire et une méthode `__del__`) sur une liste globale de déchets irrécupérables et en les laissant pour toute l'éternité:

```
/* list of uncollectable objects */
static PyObject *garbage = NULL;
```

Examples

Comptage de référence

La grande majorité de la gestion de la mémoire Python est gérée avec le comptage des références.

Chaque fois qu'un objet est référencé (par exemple, affecté à une variable), son compte de référence est automatiquement augmenté. Quand il est déréférencé (par exemple, la variable sort de la portée), son compte de référence est automatiquement diminué.

Lorsque le compteur de références atteint zéro, l'objet est **immédiatement détruit** et la mémoire est immédiatement libérée. Ainsi, dans la majorité des cas, le ramasse-miettes n'est même pas nécessaire.

```
>>> import gc; gc.disable()  # disable garbage collector
```

```
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def foo():
        Track()
        # destructed immediately since no longer has any references
        print("---")
        t = Track()
        # variable is referenced, so it's not destructed yet
        print("---")
        # variable is destructed when function exits
>>> foo()
Initialized
Destructed
---
Initialized
---
Destructed
```

Démontrer plus avant le concept de références:

```
>>> def bar():
        return Track()
>>> t = bar()
Initialized
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> t = None          # not destructed yet - another_t still refers to it
>>> another_t = None  # final reference gone, object is destructed
Destructed
```

Garbage Collector pour les cycles de référence

La seule fois où le ramasse-miettes est nécessaire, c'est si vous avez un *cycle de référence* . L'exemple simple d'un cycle de référence est celui dans lequel A désigne B et B désigne A, tandis que rien d'autre ne fait référence à A ou B. Ni A ni B ne sont accessibles de n'importe où dans le programme, ils peuvent donc être détruits en toute sécurité. cependant, leurs comptages de référence sont 1 et ils ne peuvent donc pas être libérés par l'algorithme de comptage de référence seul.

```
>>> import gc; gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> A = Track()
Initialized
>>> B = Track()
Initialized
>>> A.other = B
>>> B.other = A
>>> del A; del B  # objects are not destructed due to reference cycle
```

```
>>> gc.collect()  # trigger collection
Destructed
Destructed
4
```

Un cycle de référence peut être arbitrairement long. Si A pointe vers B pointe vers C pointe sur ... pointe vers Z qui pointe sur A, alors ni A ni Z ne seront collectés, jusqu'à la phase de récupération de place:

```
>>> objs = [Track() for _ in range(10)]
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
>>> for i in range(len(objs)-1):
...     objs[i].other = objs[i + 1]
...
>>> objs[-1].other = objs[0]  # complete the cycle
>>> del objs                  # no one can refer to objs now - still not destructed
>>> gc.collect()
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
20
```

Effets de la commande del

Suppression d'un nom de variable de l'étendue à l'aide de `del v` ou suppression d'un objet d'une collection à l'aide de `del v[item]` ou `del[i:j]` ou suppression d'un attribut avec `del v.name` ou tout autre moyen de suppression de références un objet *ne* déclenche aucun appel de destructeur ni aucune mémoire libérée en soi. Les objets ne sont détruits que lorsque leur compte de référence atteint zéro.

```
>>> import gc
>>> gc.disable()  # disable garbage collector
>>> class Track:
        def __init__(self):
            print("Initialized")
        def __del__(self):
            print("Destructed")
>>> def bar():
    return Track()
>>> t = bar()
```

```
Initialized
>>> another_t = t  # assign another reference
>>> print("...")
...
>>> del t          # not destructed yet - another_t still refers to it
>>> del another_t  # final reference gone, object is destructed
Destructed
```

Réutilisation d'objets primitifs

Une chose intéressante à noter qui peut aider à optimiser vos applications est que les primitives sont en fait également recalculées sous le capot. Jetons un coup d'oeil aux chiffres; pour tous les entiers compris entre -5 et 256, Python réutilise toujours le même objet:

```
>>> import sys
>>> sys.getrefcount(1)
797
>>> a = 1
>>> b = 1
>>> sys.getrefcount(1)
799
```

Notez que le refcount augmente, ce qui signifie que `a` et `b` référencent le même objet sous-jacent lorsqu'ils font référence à la primitive `1` . Cependant, pour les nombres plus importants, Python ne réutilise pas réellement l'objet sous-jacent:

```
>>> a = 999999999
>>> sys.getrefcount(999999999)
3
>>> b = 999999999
>>> sys.getrefcount(999999999)
3
```

Étant donné que le refcount pour `999999999` ne change pas lors de l'affectation à `a` et `b` nous pouvons en déduire qu'ils font référence à deux objets sous-jacents différents, même s'ils se voient attribuer la même primitive.

Affichage du refcount d'un objet

```
>>> import sys
>>> a = object()
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

Désallouer des objets avec force

Vous pouvez forcer les objets de désaffectation même si leur refcount n'est pas 0 dans Python 2 et 3.

Les deux versions utilisent le module `ctypes` pour le faire.

AVERTISSEMENT: cela laissera faire votre environnement Python instable et enclin à s'écraser sans retraçage! L'utilisation de cette méthode peut également créer des problèmes de sécurité (très improbable). Déjà.

Python 3.x 3.0

```
import ctypes
deallocated = 12345
ctypes.pythonapi._Py_Dealloc(ctypes.py_object(deallocated))
```

Python 2.x 2.3

```
import ctypes, sys
deallocated = 12345
(ctypes.c_char * sys.getsizeof(deallocated)).from_address(id(deallocated))[:4] = '\x00' * 4
```

Après l'exécution, toute référence à l'objet maintenant désalloué entraînera un comportement ou un blocage indéfini de Python, sans trace. Il y avait probablement une raison pour laquelle le garbage collector n'a pas supprimé cet objet ...

Si vous désallouez `None` , vous obtenez un message spécial - `Fatal Python error: deallocating None` avant de planter.

Gestion de la récupération de place

Il existe deux approches pour influencer le nettoyage de la mémoire. Ils influencent la fréquence à laquelle le processus automatique est exécuté et l'autre déclenche manuellement un nettoyage.

Le garbage collector peut être manipulé en réglant les seuils de collecte qui affectent la fréquence d'exécution du collecteur. Python utilise un système de gestion de la mémoire basé sur la génération. Les nouveaux objets sont enregistrés dans la génération la plus récente - **generation0** et avec chaque collection survivante, les objets sont promus aux générations plus anciennes. Après avoir atteint la dernière génération - la **génération 2** , ils ne sont plus promus.

Les seuils peuvent être modifiés à l'aide de l'extrait suivant:

```
import gc
gc.set_threshold(1000, 100, 10) # Values are just for demonstration purpose
```

Le premier argument représente le seuil de collecte de **génération0** . Chaque fois que le nombre d' **allocations** dépasse le nombre de **désallocations** de 1 000, le récupérateur de mémoire sera appelé.

Les générations les plus anciennes ne sont pas nettoyées à chaque exécution pour optimiser le processus. Les deuxième et troisième arguments sont **facultatifs** et contrôlent la fréquence de

nettoyage des générations les plus anciennes. Si **generation0 a** été traité 100 fois sans nettoyage de la **génération 1** , alors la **génération 1** sera traitée. De même, les objets de **génération 2** ne seront traités que lorsque ceux de **génération 1** ont été nettoyés 10 fois sans toucher à la **génération 2** .

Un exemple dans lequel la définition manuelle des seuils est bénéfique est lorsque le programme alloue un grand nombre de petits objets sans les désallouer, ce qui entraîne l'exécution trop fréquente du ramasse-miettes (chaque attribution d'objet **generation0_threshold**). Bien que le collecteur soit assez rapide, lorsqu'il fonctionne sur un grand nombre d'objets, il pose un problème de performance. Quoi qu'il en soit, il n'y a pas de stratégie unique pour choisir les seuils et son utilisation est fiable.

Le déclenchement manuel d'une collection peut se faire comme dans l'extrait de code suivant:

```
import gc
gc.collect()
```

La récupération de place est automatiquement déclenchée en fonction du nombre d'allocations et de désallocations, et non de la mémoire consommée ou disponible. Par conséquent, lorsque vous travaillez avec des objets volumineux, la mémoire peut être épuisée avant le déclenchement du nettoyage automatisé. Cela fait un bon cas d'utilisation pour appeler manuellement le garbage collector.

Même si c'est possible, ce n'est pas une pratique encouragée. Eviter les fuites de mémoire est la meilleure option. Quoi qu'il en soit, dans les grands projets, la détection de la fuite de mémoire peut être une tâche ardue et le déclenchement manuel d'une récupération de place peut être utilisé comme solution rapide jusqu'au débogage ultérieur.

Pour les programmes de longue durée, le nettoyage de la mémoire peut être déclenché sur une base ponctuelle ou par événement. Un exemple pour le premier est un serveur Web qui déclenche une collecte après un nombre fixe de requêtes. Par la suite, un serveur Web qui déclenche un nettoyage de la mémoire lorsqu'un certain type de demande est reçu.

N'attendez pas que le ramasse-miettes nettoie

Le fait que le ramasse-miettes se nettoie ne signifie pas que vous devez attendre le nettoyage du cycle de nettoyage.

En particulier, vous ne devez pas attendre le nettoyage de la mémoire pour fermer les descripteurs de fichiers, les connexions à la base de données et les connexions réseau ouvertes.

par exemple:

Dans le code suivant, vous supposez que le fichier sera fermé lors du prochain cycle de récupération de la mémoire, si f était la dernière référence au fichier.

```
>>> f = open("test.txt")
>>> del f
```

Un moyen plus explicite de nettoyer est d'appeler `f.close()` . Vous pouvez le faire encore plus élégant, en utilisant l'instruction `with` , également appelée gestionnaire de contexte :

```
>>> with open("test.txt") as f:
...     pass
...     # do something with f
>>> #now the f object still exists, but it is closed
```

L'instruction `with` vous permet d'indenter votre code sous le fichier ouvert. Cela rend explicite et plus facile de voir combien de temps un fichier reste ouvert. Il ferme également toujours un fichier, même si une exception est déclenchée dans le bloc `while` .

Lire Collecte des ordures en ligne: https://riptutorial.com/fr/python/topic/2532/collecte-des-ordures

Chapitre 24: commencer avec GZip

Introduction

Ce module fournit une interface simple pour compresser et décompresser des fichiers comme le feraient les programmes GNU gzip et gunzip.

La compression des données est fournie par le module zlib.

Le module gzip fournit la classe GzipFile qui est modélisée après l'objet File de Python. La classe GzipFile lit et écrit les fichiers au format gzip, compressant ou décompressant automatiquement les données pour qu'elles ressemblent à un objet fichier ordinaire.

Examples

Lire et écrire des fichiers zip GNU

```
import gzip
import os

outfilename = 'example.txt.gz'
output = gzip.open(outfilename, 'wb')
try:
    output.write('Contents of the example file go here.\n')
finally:
    output.close()

print outfilename, 'contains', os.stat(outfilename).st_size, 'bytes of compressed data'
os.system('file -b --mime %s' % outfilename)
```

Enregistrez-le sous le nom 1gzip_write.py1.Exécutez-le via le terminal.

```
$ python gzip_write.py

application/x-gzip; charset=binary
example.txt.gz contains 68 bytes of compressed data
```

Lire commencer avec GZip en ligne: https://riptutorial.com/fr/python/topic/8993/commencer-avec-gzip

Chapitre 25: Commentaires et documentation

Syntaxe

- # Ceci est un commentaire sur une seule ligne
- print ("") # Ceci est un commentaire en ligne
- "" "
  C'est
  un commentaire sur plusieurs lignes
  "" "

Remarques

Les développeurs doivent suivre les directives PEP257 - Conventions Docstring . Dans certains cas, des guides de style (tels que ceux de Google Style Guide) ou des tiers de rendu de documentation (tels que Sphinx) peuvent détailler des conventions supplémentaires pour docstrings.

Examples

Commentaires sur une seule ligne, inline et multiligne

Les commentaires sont utilisés pour expliquer le code lorsque le code de base lui-même n'est pas clair.

Python ignore les commentaires et n'exécute donc pas de code là-dedans, ni ne génère d'erreurs de syntaxe pour les phrases anglaises simples.

Les commentaires sur une seule ligne commencent par le caractère de hachage (`#`) et se terminent par la fin de la ligne.

- Commentaire sur une seule ligne:

```
# This is a single line comment in Python
```

- Commentaire en ligne:

```
print("Hello World")  # This line prints "Hello World"
```

- Les commentaires couvrant plusieurs lignes ont `"""` ou `'''` à chaque extrémité. C'est la même chose qu'une chaîne multiligne, mais ils peuvent être utilisés comme commentaires:

```
"""
This type of comment spans multiple lines.
These are mostly used for documentation of functions, classes and modules.
```

```
"""
```

Accéder par programme à docstrings

Contrairement aux commentaires réguliers, Docstrings est stocké en tant qu'attribut de la fonction qu'ils documentent, ce qui signifie que vous pouvez y accéder par programmation.

Un exemple de fonction

```
def func():
    """This is a function that does nothing at all"""
    return
```

Le docstring est accessible en utilisant l'attribut `__doc__` :

```
print(func.__doc__)
```

```
    Ceci est une fonction qui ne fait rien du tout
```

```
help(func)
```

```
    Aide sur la fonction func dans le module __main__ :

    func()

    Ceci est une fonction qui ne fait rien du tout
```

Un autre exemple de fonction

`function.__doc__` n'est que le docstring réel en tant que chaîne, alors que la fonction `help` fournit des informations générales sur une fonction, y compris la docstring. Voici un exemple plus utile:

```
def greet(name, greeting="Hello"):
    """Print a greeting to the user `name`

    Optional parameter `greeting` can change what they're greeted with."""

    print("{} {}".format(greeting, name))
```

```
help(greet)
```

```
    Aide sur la fonction greet dans le module __main__ :

    greet(name, greeting='Hello')

    Imprimer un message d'accueil au name de l'utilisateur
    Paramètre optionnel greeting d' greeting peut changer ce qu'ils sont accueillis avec.
```

Avantages de docstrings sur les commentaires réguliers

Le simple fait de ne pas placer de docstring ou un commentaire régulier dans une fonction le rend moins utile.

```
def greet(name, greeting="Hello"):
    # Print a greeting to the user `name`
    # Optional parameter `greeting` can change what they're greeted with.

    print("{} {}".format(greeting, name))
```

```
print(greet.__doc__)
```

Aucun

```
help(greet)
```

Aide sur la fonction accueil dans le module **principal** :

```
greet(name, greeting='Hello')
```

Ecrire de la documentation à l'aide de docstrings

Un docstring est un commentaire multi-lignes utilisé pour documenter des modules, des classes, des fonctions et des méthodes. Il doit s'agir de la première déclaration du composant qu'il décrit.

```
def hello(name):
    """Greet someone.

    Print a greeting ("Hello") for the person with the given name.
    """

    print("Hello "+name)
```

```
class Greeter:
    """An object used to greet people.

    It contains multiple greeting functions for several languages
    and times of the  day.
    """
```

La valeur de docstring est accessible dans le programme et est, par exemple, utilisée par la commande `help` .

Conventions de syntaxe

PEP 257

PEP 257 définit une norme de syntaxe pour les commentaires docstring. Il permet essentiellement

deux types:

- Docstrings à une ligne:

Selon le PEP 257, ils doivent être utilisés avec des fonctions courtes et simples. Tout est placé sur une ligne, par exemple:

```
def hello():
    """Say hello to your friends."""
    print("Hello my friends!")
```

Le docstring doit se terminer par un point, le verbe doit être sous la forme impérative.

- Docstrings multilignes:

Les docstrings multi-lignes doivent être utilisés pour des fonctions, modules ou classes plus longs et plus complexes.

```
def hello(name, language="en"):
    """Say hello to a person.

    Arguments:
    name: the name of the person
    language: the language in which the person should be greeted
    """

    print(greeting[language]+" "+name)
```

Ils commencent par un bref résumé (équivalent au contenu d'une docstring à une ligne) qui peut être sur la même ligne que les guillemets ou sur la ligne suivante, donnant des détails supplémentaires, des paramètres de liste et des valeurs de retour.

Remarque PEP 257 définit quelles informations doivent être données dans un document, il ne définit pas dans quel format il doit être donné. C'était la raison pour laquelle d'autres parties et des outils d'analyse de documentation spécifiaient leurs propres normes pour la documentation, dont certaines sont énumérées ci-dessous et dans cette question .

Sphinx

Sphinx est un outil permettant de générer de la documentation basée sur HTML pour les projets Python basés sur docstrings. Son langage de balisage utilisé est reStructuredText . Ils définissent leurs propres normes pour la documentation, pythonhosted.org en héberge une très bonne description . Le format Sphinx est par exemple utilisé par l' IDE pyCharm .

Une fonction serait documentée comme ceci en utilisant le format Sphinx / reStructuredText:

```
def hello(name, language="en"):
    """Say hello to a person.

    :param name: the name of the person
    :type name: str
```

```
    :param language: the language in which the person should be greeted
    :type language: str
    :return: a number
    :rtype: int
    """

    print(greeting[language]+" "+name)
    return 4
```

Guide de style Google Python

Google a publié le guide de style Google Python qui définit les conventions de codage pour Python, y compris les commentaires de la documentation. En comparaison avec Sphinx / reST, beaucoup de gens disent que la documentation selon les directives de Google est mieux lisible par l'homme.

La page pythonhosted.org mentionnée ci-dessus fournit également des exemples de documentation conforme au Guide de style de Google.

En utilisant le plug-in Napoleon , Sphinx peut également analyser la documentation dans le format compatible avec Google Style Guide.

Une fonction serait documentée comme ceci en utilisant le format Guide de style de Google:

```
def hello(name, language="en"):
    """Say hello to a person.

    Args:
        name: the name of the person as string
        language: the language code string

    Returns:
        A number.
    """

    print(greeting[language]+" "+name)
    return 4
```

Lire Commentaires et documentation en ligne:
https://riptutorial.com/fr/python/topic/4144/commentaires-et-documentation

Chapitre 26: Communication série Python (pyserial)

Syntaxe

- ser.read (taille = 1)
- ser.readline ()
- ser.write ()

Paramètres

paramètre	détails
Port	Nom du périphérique, par exemple / dev / ttyUSB0 sur GNU / Linux ou COM3 sur Windows.
baudrate	baudrate type: int par défaut: 9600 valeurs standard: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200

Remarques

Pour plus de détails, consultez la documentation pyserial

Examples

Initialiser le périphérique série

```
import serial
#Serial takes these two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

Lire depuis le port série

Initialiser le périphérique série

```
import serial
#Serial takes two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

lire un seul octet à partir d'un périphérique série

```
data = ser.read()
```

lire le nombre donné d'octets du périphérique série

```
data = ser.read(size=5)
```

lire une ligne du périphérique série.

```
data = ser.readline()
```

lire les données du périphérique série pendant que quelque chose est écrit dessus.

```
#for python2.7
data = ser.read(ser.inWaiting())

#for python3
ser.read(ser.inWaiting)
```

Vérifiez quels ports série sont disponibles sur votre machine

Pour obtenir une liste des ports série disponibles, utilisez

```
python -m serial.tools.list_ports
```

à une invite de commande ou

```
from serial.tools import list_ports
list_ports.comports()  # Outputs list of available serial ports
```

à partir du shell Python.

Lire Communication série Python (pyserial) en ligne:
https://riptutorial.com/fr/python/topic/5744/communication-serie-python--pyserial-

Chapitre 27: Comparaisons

Syntaxe

- ! = - n'est pas égal à
- == - Est égal à
- `>` - supérieur à
- `<` - moins que
- `>=` - supérieur ou égal à
- `<=` - inférieur ou égal à
- `is` - teste si les objets sont exactement le même objet
- is not = test si les objets ne sont pas exactement le même objet

Paramètres

Paramètre	Détails
X	Premier article à comparer
y	Deuxième élément à comparer

Examples

Supérieur ou inférieur à

```
x > y
x < y
```

Ces opérateurs comparent deux types de valeurs, ils sont inférieurs et supérieurs aux opérateurs. Pour les nombres, il suffit de comparer les valeurs numériques pour voir laquelle est la plus grande:

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

Pour les chaînes, ils comparent lexicographiquement, ce qui est similaire à l'ordre alphabétique mais pas tout à fait pareil.

```
"alpha" < "beta"
# True
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

Dans ces comparaisons, les lettres minuscules sont considérées comme «supérieures à» les majuscules, ce qui explique pourquoi `"gamma" < "OMEGA"` est faux. S'ils étaient tous en majuscules, ils renverraient le résultat de l'ordre alphabétique attendu:

```
"GAMMA" < "OMEGA"
# True
```

Chaque type définit son calcul avec les opérateurs `<` et `>` différemment, vous devez donc étudier ce que les opérateurs veulent dire avec un type donné avant de l'utiliser.

Pas égal à

```
x != y
```

Cela renvoie `True` si `x` et `y` ne sont pas égaux et renvoie sinon `False` .

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

Égal à

```
x == y
```

Cette expression évalue si `x` et `y` sont la même valeur et renvoie le résultat sous forme de valeur booléenne. Généralement, le type et la valeur doivent correspondre, donc l'int `12` n'est pas la même que la chaîne `'12'` .

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
# True
'spam' == 'spam '
# False
```

```
'12' == 12
# False
```

Notez que chaque type doit définir une fonction qui sera utilisée pour évaluer si deux valeurs sont identiques. Pour les types intégrés, ces fonctions se comportent comme prévu et évaluent simplement les choses en fonction de la même valeur. Cependant, les types personnalisés peuvent définir le test d'égalité comme ils le souhaitent, notamment en renvoyant toujours `True` ou en retournant toujours `False` .

Comparaisons en chaîne

Vous pouvez comparer plusieurs éléments avec plusieurs opérateurs de comparaison avec une comparaison en chaîne. Par exemple

```
x > y > z
```

est juste une forme abrégée de:

```
x > y and y > z
```

Ceci évaluera à `True` seulement si les deux comparaisons sont `True` .

La forme générale est

```
a OP b OP c OP d ...
```

Où `OP` représente l'une des multiples opérations de comparaison que vous pouvez utiliser, et les lettres représentent des expressions valides arbitraires.

> Notez que `0 != 1 != 0 0 != 0` vaut `True` , même si `0 != 0` est `False` . Contrairement à la notation mathématique commune dans laquelle `x != y != z` signifie que `x` , `y` et `z` ont des valeurs différentes. Le chaînage des opérations `==` a un sens naturel dans la plupart des cas, puisque l'égalité est généralement transitive.

Style

Il n'y a pas de limite théorique sur le nombre d'éléments et d'opérations de comparaison que vous utilisez tant que vous avez la syntaxe appropriée:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

Ce qui précède renvoie `True` si chaque comparaison renvoie `True` . Cependant, utiliser un chaînage compliqué n'est pas un bon style. Un bon chaînage sera "directionnel", pas plus compliqué que

```
1 > x > -4 > y != 8
```

Effets secondaires

Dès qu'une comparaison renvoie `False` , l'expression est immédiatement évaluée à `False` , en ignorant toutes les comparaisons restantes.

Notez que l'expression `exp` dans `a > exp > b` sera évaluée une seule fois, alors que dans le cas de

```
a > exp and exp > b
```

`exp` sera calculée deux fois si `a > exp` est vraie.

Comparaison par `is` vs` == `

Un écueil courant confond les opérateurs de comparaison de l' égalité `is` et `==` .

`a == b` compare la valeur de `a` et `b` .

`a is b` comparera les *identités* de `a` et `b` .

Pour illustrer:

```
a = 'Python is fun!'
b = 'Python is fun!'
a == b # returns True
a is b # returns False

a = [1, 2, 3, 4, 5]
b = a      # b references a
a == b     # True
a is b     # True
b = a[:]   # b now references a copy of a
a == b     # True
a is b     # False [!!]
```

Fondamentalement, `is` peut être considéré comme un raccourci pour `id(a) == id(b)` .

Au-delà de cela, il existe des particularités de l'environnement d'exécution qui compliquent davantage les choses. Les chaînes courtes et les petits entiers renvoient `True` par rapport à `is` , car la machine Python tente d'utiliser moins de mémoire pour des objets identiques.

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

Mais les chaînes plus longues et les entiers plus grands seront stockés séparément.

```
a = 'not so short'
b = 'not so short'
```

```
c = 1000
d = 1000
a is b # False
c is d # False
```

Vous devez utiliser `is` de tester pour `None` :

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

Une utilisation de `is` de tester un "sentinelle" (c'est-à-dire un objet unique).

```
sentinel = object()
def myfunc(var=sentinel):
    if var is sentinel:
        # value wasn't provided
        pass
    else:
        # value was provided
        pass
```

Comparer des objets

Pour comparer l'égalité des classes personnalisées, vous pouvez remplacer `==` et `!=` définissant les méthodes `__eq__` et `__ne__` . Vous pouvez également remplacer `__lt__` (`<`), `__le__` (`<=`), `__gt__` (`>`) et `__ge__` (`>`). Notez que vous devez uniquement remplacer deux méthodes de comparaison, et Python peut gérer le reste (`==` est le même que `not <` et `not >` , etc.)

```
class Foo(object):
    def __init__(self, item):
        self.my_item = item
    def __eq__(self, other):
        return self.my_item == other.my_item

a = Foo(5)
b = Foo(5)
a == b     # True
a != b     # False
a is b     # False
```

Notez que cette simple comparaison suppose que l' `other` (l'objet comparé à) est le même type d'objet. La comparaison avec un autre type provoquera une erreur:

```
class Bar(object):
    def __init__(self, item):
        self.other_item = item
    def __eq__(self, other):
        return self.other_item == other.other_item
    def __ne__(self, other):
        return self.other_item != other.other_item
```

```
c = Bar(5)
a == c    # throws AttributeError: 'Foo' object has no attribute 'other_item'
```

Vérifier `isinstance()` ou similaire aidera à empêcher cela (si désiré).

Common Gotcha: Python n'impose pas la saisie

Dans beaucoup d'autres langues, si vous exécutez ce qui suit (exemple Java)

```
if("asgdsrf" == 0) {
    //do stuff
}
```

... vous aurez une erreur. Vous ne pouvez pas simplement comparer des chaînes à des entiers comme celui-là. En Python, il s'agit d'une déclaration parfaitement légale - elle se résume à `False` .

Un gotcha commun est le suivant

```
myVariable = "1"
if 1 == myVariable:
    #do stuff
```

Cette comparaison sera évaluée à `False` sans erreur, à chaque fois, masquant potentiellement un bogue ou interrompant une condition.

Lire Comparaisons en ligne: https://riptutorial.com/fr/python/topic/248/comparaisons

Chapitre 28: Compte

Examples

Compter toutes les occurrences de tous les éléments dans un iterable: collection.Counter

```
from collections import Counter

c = Counter(["a", "b", "c", "d", "a", "b", "a", "c", "d"])
c
# Out: Counter({'a': 3, 'b': 2, 'c': 2, 'd': 2})
c["a"]
# Out: 3

c[7]     # not in the list (7 occurred 0 times!)
# Out: 0
```

Les `collections.Counter` peuvent être utilisées pour toute itération et compte chaque occurrence pour chaque élément.

Note : Une exception est si un `dict` ou une autre `collections.Mapping` classe-like est donnée, alors elle ne les comptera pas, mais crée un compteur avec ces valeurs:

```
Counter({"e": 2})
# Out: Counter({"e": 2})

Counter({"e": "e"})        # warning Counter does not verify the values are int
# Out: Counter({"e": "e"})
```

Obtenir la valeur la plus courante (-s): collections.Counter.most_common ()

Compter les *clés* d'un `Mapping` n'est pas possible avec `collections.Counter` mais nous pouvons compter les *valeurs* :

```
from collections import Counter
adict = {'a': 5, 'b': 3, 'c': 5, 'd': 2, 'e':2, 'q': 5}
Counter(adict.values())
# Out: Counter({2: 2, 3: 1, 5: 3})
```

Les éléments les plus courants sont disponibles par la `most_common` most_common:

```
# Sorting them from most-common to least-common value:
Counter(adict.values()).most_common()
# Out: [(5, 3), (2, 2), (3, 1)]

# Getting the most common value
Counter(adict.values()).most_common(1)
# Out: [(5, 3)]
```

```
# Getting the two most common values
Counter(adict.values()).most_common(2)
# Out: [(5, 3), (2, 2)]
```

Compter les occurrences d'un élément dans une séquence: list.count () et tuple.count ()

```
alist = [1, 2, 3, 4, 1, 2, 1, 3, 4]
alist.count(1)
# Out: 3

atuple = ('bear', 'weasel', 'bear', 'frog')
atuple.count('bear')
# Out: 2
atuple.count('fox')
# Out: 0
```

Compter les occurrences d'une sous-chaîne dans une chaîne: str.count ()

```
astring = 'thisisashorttext'
astring.count('t')
# Out: 4
```

Cela fonctionne même pour des sous-chaînes de plus d'un caractère:

```
astring.count('th')
# Out: 1
astring.count('is')
# Out: 2
astring.count('text')
# Out: 1
```

ce qui ne serait pas possible avec des `collections.Counter` qui ne compte que des caractères simples:

```
from collections import Counter
Counter(astring)
# Out: Counter({'a': 1, 'e': 1, 'h': 2, 'i': 2, 'o': 1, 'r': 1, 's': 3, 't': 4, 'x': 1})
```

Comptage des occurrences dans le tableau numpy

Pour compter les occurrences d'une valeur dans un tableau numpy. Cela fonctionnera:

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

La logique est que l'instruction booléenne produit un tableau où toutes les occurrences des valeurs demandées sont 1 et toutes les autres sont zéro. Donc, en les additionnant, vous obtenez

le nombre d'occurrences. Cela fonctionne pour les tableaux de toute forme ou dtype.

Il existe deux méthodes pour compter les occurrences de toutes les valeurs uniques dans numpy. Unique et montant. Unique aplatit automatiquement les tableaux multidimensionnels, tandis que bincount ne fonctionne qu'avec les tableaux 1d contenant uniquement des entiers positifs.

```
>>> unique,counts=np.unique(a,return_counts=True)
>>> print unique,counts # counts[i] is equal to occurrences of unique[i] in a
[0 3 4 5 7] [1 2 2 1 1]
>>> bin_count=np.bincount(a)
>>> print bin_count # bin_count[i] is equal to occurrences of i in a
[1 0 0 2 2 1 0 1]
```

Si vos données sont des tableaux numpy, il est généralement beaucoup plus rapide d'utiliser les méthodes numpy que de convertir vos données en méthodes génériques.

Lire Compte en ligne: https://riptutorial.com/fr/python/topic/476/compte

Chapitre 29: Concurrence Python

Remarques

Les développeurs Python se sont assurés que l'API entre les `threading` et le `multiprocessing` est similaire, de sorte que la commutation entre les deux variantes est plus facile pour les programmeurs.

Examples

Le module de filetage

```
from __future__ import print_function
import threading
def counter(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
t2 = threading.Thread(target=countdown,args=(20,))
t2.start()
```

Dans certaines implémentations de Python tels que CPython, le vrai parallélisme n'est pas réalisée à l' aide des fils en raison de l' utilisation de ce qui est connu comme le GIL, ou **G** Lobal **I** nterpreter **L** ock.

Voici un excellent aperçu de la concurrence Python:

La concurrence Python par David Beazley (YouTube)

Le module de multitraitement

```
from __future__ import print_function
import multiprocessing


def countdown(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=countdown, args=(10,))
    p1.start()

    p2 = multiprocessing.Process(target=countdown, args=(20,))
    p2.start()
```

```
    p1.join()
    p2.join()
```

Ici, chaque fonction est exécutée dans un nouveau processus. Comme une nouvelle instance de Python VM exécute le code, il n'y a pas de `GIL` et le parallélisme s'exécute sur plusieurs cœurs.

La méthode `Process.start` lance ce nouveau processus et exécute la fonction passée dans l'argument `target` avec les arguments `args` . La méthode `Process.join` attend la fin de l'exécution des processus `p1` et `p2` .

Les nouveaux processus sont lancés différemment selon la version de python et la plateforme sur laquelle le code est exécuté, *par exemple* :

- Windows utilise `spawn` pour créer le nouveau processus.
- Avec les systèmes unix et la version antérieure à la version 3.3, les processus sont créés à l'aide d'un `fork` .
  Notez que cette méthode ne respecte pas l'utilisation POSIX de fork et conduit donc à des comportements inattendus, notamment lors de l'interaction avec d'autres bibliothèques multiprocesseurs.
- Avec le système unix et la version 3.4+, vous pouvez choisir de lancer les nouveaux processus avec `fork` , `forkserver` ou `spawn` utilisant `multiprocessing.set_start_method` au début de votre programme. `forkserver` méthodes de `forkserver` et de `spawn` sont plus lentes que les méthodes de forking mais évitent certains comportements inattendus.

Utilisation de la fourche POSIX :

> Après un fork dans un programme multithread, l'enfant peut en toute sécurité appeler uniquement les fonctions async-signal-safe jusqu'à ce qu'il appelle execve.
> (voir)

En utilisant fork, un nouveau processus sera lancé avec exactement le même état pour tous les mutex actuels, mais seul le `MainThread` sera lancé. Ceci est dangereux car cela pourrait conduire à des conditions de course, *par exemple* :

- Si vous utilisez un `Lock` dans `MainThread` et le transmettez à un autre thread qui est censé le verrouiller à un moment donné. Si le `fork` simultanément, le nouveau processus démarrera avec un verrou verrouillé qui ne sera jamais publié car le deuxième thread n'existe pas dans ce nouveau processus.

En fait, ce type de comportement ne devrait pas se produire en python pur, car le `multiprocessing` gère correctement, mais si vous interagissez avec d'autres bibliothèques, ce type de comportement peut entraîner un blocage de votre système (par exemple, numpy / accéléré sur macOS).

Passer des données entre des processus multiprocessus

Les données étant sensibles lorsqu'elles sont traitées entre deux threads (pensez que les lectures simultanées et les écritures concurrentes peuvent être en conflit les unes avec les autres,

provoquant des conditions de concurrence), un ensemble d'objets uniques a été créé pour faciliter la transmission des données entre les threads. Toute opération vraiment atomique peut être utilisée entre les threads, mais il est toujours prudent de s'en tenir à la file d'attente.

```
import multiprocessing
import queue
my_Queue=multiprocessing.Queue()
#Creates a queue with an undefined maximum size
#this can be dangerous as the queue becomes increasingly large
#it will take a long time to copy data to/from each read/write thread
```

Lors de l'utilisation de la file d'attente, la plupart des utilisateurs suggèrent de toujours placer les données de la file d'attente dans un essai: sauf: bloquer au lieu d'utiliser vide. Cependant, pour les applications où il est `queue.Empty==True` passer un cycle d'analyse (les données peuvent être placées dans la file d'attente en retournant les états de `queue.Empty==True` en `queue.Empty==False` d' `queue.Empty==False`) il est généralement préférable de lire et écrivez un accès dans ce que j'appelle un bloc Iftry, car une instruction "if" est techniquement plus performante que la capture d'une exception.

```
import multiprocessing
import queue
'''Import necessary Python standard libraries, multiprocessing for classes and queue for the
queue exceptions it provides'''
def Queue_Iftry_Get(get_queue, default=None, use_default=False, func=None, use_func=False):
    '''This global method for the Iftry block is provided for it's reuse and
standard functionality, the if also saves on performance as opposed to catching
 the exception, which is expencive.
        It also allows the user to specify a function for the outgoing data to use,
 and a default value to return if the function cannot return the value from the queue'''
        if get_queue.empty():
            if use_default:
                return default
        else:
            try:
                value = get_queue.get_nowait()
            except queue.Empty:
                if use_default:
                    return default
            else:
                if use_func:
                    return func(value)
                else:
                    return value
    def Queue_Iftry_Put(put_queue, value):
        '''This global method for the Iftry block is provided because of its reuse
and
standard functionality, the If also saves on performance as opposed to catching
 the exception, which is expensive.
        Return True if placing value in the queue was successful. Otherwise, false'''
        if put_queue.full():
            return False
        else:
            try:
                put_queue.put_nowait(value)
            except queue.Full:
                return False
            else:
```

```
                return True
```

Lire Concurrence Python en ligne: https://riptutorial.com/fr/python/topic/3357/concurrence-python

Chapitre 30: Conditionnels

Introduction

Les expressions conditionnelles, impliquant des mots-clés tels que if, elif et bien d'autres, permettent aux programmes Python d'effectuer différentes actions en fonction d'une condition booléenne: True ou False. Cette section traite de l'utilisation des conditions Python, de la logique booléenne et des instructions ternaires.

Syntaxe

- <expression> if <conditionnel> else <expression> # Opérateur ternaire

Examples

si elif et autre

En Python, vous pouvez définir une série de conditions en utilisant `if` pour le premier, `elif` pour le reste, jusqu'au final (optionnel) `else` pour tout ce qui n'est pas pris par les autres conditionnelles.

```
number = 5

if number > 2:
    print("Number is bigger than 2.")
elif number < 2:  # Optional clause (you can have multiple elifs)
    print("Number is smaller than 2.")
else:  # Optional clause (you can only have one else)
    print("Number is 2.")
```

Le `Number is bigger than 2` sorties `Number is bigger than 2`

Utiliser `else if` au lieu de `elif` déclenchera une erreur de syntaxe et n'est pas autorisé.

Expression conditionnelle (ou "l'opérateur ternaire")

L'opérateur ternaire est utilisé pour les expressions conditionnelles en ligne. Il est préférable de l'utiliser dans des opérations simples et concises, faciles à lire.

- L'ordre des arguments est différent de beaucoup d'autres langages (tels que C, Ruby, Java, etc.), ce qui peut conduire à des bogues lorsque des personnes peu familiarisées avec le comportement "surprenant" de Python l'utilisent (elles peuvent inverser l'ordre).
- Certains le trouvent "peu maniable", car cela va à l'encontre du flux normal de la pensée (penser d'abord à la condition et ensuite aux effets).

```
n = 5
```

```
"Greater than 2" if n > 2 else "Smaller than or equal to 2"
# Out: 'Greater than 2'
```

Le résultat de cette expression sera tel qu'il est lu en anglais - si l'expression conditionnelle est True, alors elle sera évaluée à l'expression du côté gauche, sinon du côté droit.

Les opérations de tenaire peuvent également être imbriquées, comme ici:

```
n = 5
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

Ils fournissent également une méthode pour inclure des conditions dans les fonctions lambda .

Si déclaration

```
if condition:
    body
```

Les instructions `if` vérifient la condition. S'il est évalué à `True` , il exécute le corps de l'instruction `if` . S'il est évalué à `False` , il saute le corps.

```
if True:
    print "It is true!"
>> It is true!

if False:
    print "This won't get printed.."
```

La condition peut être toute expression valide:

```
if 2 + 2 == 4:
    print "I know math!"
>> I know math!
```

Autre déclaration

```
if condition:
    body
else:
    body
```

L'instruction else n'exécutera son corps que si les instructions conditionnelles précédentes valent toutes False.

```
if True:
    print "It is true!"
else:
    print "This won't get printed.."

# Output: It is true!
```

```
if False:
    print "This won't get printed.."
else:
    print "It is false!"

# Output: It is false!
```

Expressions booléennes

Les expressions logiques booléennes, en plus d'évaluer `True` ou `False` , renvoient la *valeur* interprétée comme `True` ou `False` . C'est une manière pythonique de représenter une logique qui pourrait autrement nécessiter un test if-else.

Et opérateur

L'opérateur `and` évalue toutes les expressions et renvoie la dernière expression si toutes les expressions ont la valeur `True` . Sinon, il renvoie la première valeur évaluée à `False` :

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

Ou opérateur

L'opérateur `or` évalue les expressions de gauche à droite et retourne la première valeur qui a la valeur `True` ou la dernière valeur (si aucune n'est `True`).

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

Évaluation paresseuse

Lorsque vous utilisez cette approche, rappelez-vous que l'évaluation est paresseuse. Les expressions qui ne doivent pas être évaluées pour déterminer le résultat ne sont pas évaluées. Par exemple:

```
>>> def print_me():
        print('I am here!')
>>> 0 and print_me()
0
```

Dans l'exemple ci-dessus, `print_me` n'est jamais exécuté car Python peut déterminer que l'expression entière est `False` lorsqu'elle rencontre le `0` (`False`). Gardez cela à l'esprit si `print_me` doit être exécuté pour servir la logique de votre programme.

Test pour plusieurs conditions

Une erreur courante lors de la vérification de plusieurs conditions consiste à appliquer la logique de manière incorrecte.

Cet exemple essaie de vérifier si deux variables sont chacune supérieures à 2. L'instruction est évaluée comme suit - `if (a) and (b > 2)` . Cela produit un résultat inattendu car `bool(a)` évalué comme `True` lorsque `a` n'est pas nul.

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')

yes
```

Chaque variable doit être comparée séparément.

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')

no
```

Une autre erreur, similaire, est commise lors de la vérification d'une variable parmi plusieurs. L'énoncé dans cet exemple est évalué comme - `if (a == 3) or (4) or (6)` . Cela produit un résultat inattendu car `bool(4)` et `bool(6)` sont évalués à `True`

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
...     print('no')
```

```
yes
```

Encore une fois, chaque comparaison doit être faite séparément

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')

no
```

Utiliser l'opérateur in est la manière canonique d'écrire ceci.

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')

no
```

Valeurs de vérité

Les valeurs suivantes sont considérées comme fausses, en ce sens qu'elles sont évaluées à `False` lorsqu'elles sont appliquées à un opérateur booléen.

- Aucun
- Faux
- `0` , ou toute valeur numérique équivalente à zéro, par exemple `0L` , `0.0` , `0j`
- Séquences vides: `''` , `""` , `()` , `[]`
- Mappages vides: `{}`
- Types définis par l'utilisateur où les méthodes `__bool__` ou `__len__` renvoient 0 ou `False`

Toutes les autres valeurs de Python sont évaluées à `True` .

Remarque: une erreur courante consiste à vérifier simplement la fausseté d'une opération qui renvoie des valeurs Falsey différentes lorsque la différence est importante. Par exemple, utiliser `if foo()` plutôt que le plus explicite `if foo() is None`

Utiliser la fonction cmp pour obtenir le résultat de la comparaison de deux objets

Python 2 inclut une fonction `cmp` qui vous permet de déterminer si un objet est inférieur, égal ou supérieur à un autre objet. Cette fonction peut être utilisée pour choisir un choix parmi une liste en fonction de l'une de ces trois options.

Supposons que vous ayez besoin d'imprimer `'greater than'` si `x > y` , `'less than'` si `x < y` et `'equal'` si `x == y` .

```
['equal', 'greater than', 'less than', ][cmp(x,y)]

# x,y = 1,1 output: 'equal'
# x,y = 1,2 output: 'less than'
# x,y = 2,1 output: 'greater than'
```

`cmp(x,y)` renvoie les valeurs suivantes

Comparaison	Résultat
x <y	-1
x == y	0
x> y	1

Cette fonction est supprimée sur Python 3. Vous pouvez utiliser la `cmp_to_key(func)` assistance `cmp_to_key(func)` située dans `functools` dans Python 3 pour convertir les anciennes fonctions de comparaison en fonctions clés.

Évaluation des expressions conditionnelles à l'aide de listes de compréhension

Python vous permet de pirater les listes de compréhension pour évaluer les expressions conditionnelles.

Par exemple,

```
[value_false, value_true][<conditional-test>]
```

Exemple:

```
>> n = 16
>> print [10, 20][n <= 15]
10
```

Ici `n<=15` renvoie `False` (ce qui équivaut à 0 en Python). Donc, ce que Python évalue est:

```
[10, 20][n <= 15]
==> [10, 20][False]
==> [10, 20][0]     #False==0, True==1 (Check Boolean Equivalencies in Python)
==> 10
```

Python 2.x 2.7

La méthode intégrée `__cmp__` renvoyé 3 valeurs possibles: 0, 1, -1, où cmp (x, y) a renvoyé 0: si les deux objectifs étaient identiques 1: x> y -1: x <y

Cela pourrait être utilisé avec des listes compréhensibles pour renvoyer le premier élément (c'est-à-dire index 0), le deuxième élément (ie index 1) et le dernier élément (ie index -1) de la liste.

Nous donner un conditionnel de ce type:

```
[value_equals, value_greater, value_less][<conditional-test>]
```

Enfin, dans tous les exemples ci-dessus, Python évalue les deux branches avant d'en choisir une. Pour évaluer uniquement la branche choisie:

```
[lambda: value_false, lambda: value_true][<test>]()
```

où l'ajout de `()` à la fin garantit que les fonctions lambda ne sont appelées / évaluées qu'à la fin. Ainsi, nous n'évaluons que la branche choisie.

Exemple:

```
count = [lambda:0, lambda:N+1][count==N]()
```

Tester si un objet est Aucun et l'attribuer

Vous voudrez souvent attribuer quelque chose à un objet s'il est `None` , indiquant qu'il n'a pas été affecté. Nous utiliserons `aDate` .

La méthode la plus simple consiste à utiliser le test `is None` .

```
if aDate is None:
    aDate=datetime.date.today()
```

(Notez qu'il est plus pythonique de dire que `is None` au lieu de `== None` .)

Mais cela peut être légèrement optimisé en exploitant la notion que `not None` sera évalué à `True` dans une expression booléenne. Le code suivant est équivalent:

```
if not aDate:
    aDate=datetime.date.today()
```

Mais il y a une manière plus pythonique. Le code suivant est également équivalent:

```
aDate=aDate or datetime.date.today()
```

Cela fait une évaluation de court-circuit . Si `aDate` est initialisé et n'est `not None` , il se lui attribue sans effet net. S'il `is None` , le `datetime.date.today()` est assigné à `aDate` .

Lire Conditionnels en ligne: https://riptutorial.com/fr/python/topic/1111/conditionnels

Chapitre 31: configparser

Introduction

Ce module fournit la classe ConfigParser qui implémente un langage de configuration de base dans les fichiers INI. Vous pouvez l'utiliser pour écrire des programmes Python pouvant être personnalisés par les utilisateurs finaux.

Syntaxe

- Chaque nouvelle ligne contient une nouvelle paire de valeurs de clé séparées par le signe =
- Les clés peuvent être séparées en sections
- Dans le fichier INI, chaque titre de section est écrit entre parenthèses: []

Remarques

Toutes les valeurs `ConfigParser.ConfigParser().get` par `ConfigParser.ConfigParser().get` sont des chaînes. Il peut être converti en types plus courants grâce à `eval`

Examples

Utilisation de base

Dans config.ini:

```
[DEFAULT]
debug = True
name = Test
password = password

[FILES]
path = /path/to/file
```

En Python:

```
from ConfigParser import ConfigParser
config = ConfigParser()

#Load configuration file
config.read("config.ini")

# Access the key "debug" in "DEFAULT" section
config.get("DEFAULT", "debug")
# Return 'True'

# Access the key "path" in "FILES" destion
config.get("FILES", "path")
# Return '/path/to/file'
```

Créer un fichier de configuration par programmation

Le fichier de configuration contient des sections, chaque section contient des clés et des valeurs. Le module configparser peut être utilisé pour lire et écrire des fichiers de configuration. Création du fichier de configuration: -

```
import configparser
config = configparser.ConfigParser()
config['settings']={'resolution':'320x240',
                    'color':'blue'}
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

Le fichier de sortie contient la structure ci-dessous

```
[settings]
resolution = 320x240
color = blue
```

Si vous souhaitez modifier un champ particulier, récupérez le champ et affectez la valeur

```
settings=config['settings']
settings['color']='red'
```

Lire configparser en ligne: https://riptutorial.com/fr/python/topic/9186/configparser

Chapitre 32: Connexion de Python à SQL Server

Examples

Se connecter au serveur, créer une table, données de requête

Installez le paquet:

```
$ pip install pymssql
```

```
import pymssql

SERVER = "servername"
USER = "username"
PASSWORD = "password"
DATABASE = "dbname"

connection = pymssql.connect(server=SERVER, user=USER,
                password=PASSWORD, database=DATABASE)

cursor = connection.cursor() # to access field as dictionary use cursor(as_dict=True)
cursor.execute("SELECT TOP 1 * FROM TableName")
row = cursor.fetchone()

######## CREATE TABLE ########
cursor.execute("""
CREATE TABLE posts (
    post_id INT PRIMARY KEY NOT NULL,
    message TEXT,
    publish_date DATETIME
)
""")

######## INSERT DATA IN TABLE ########
cursor.execute("""
    INSERT INTO posts VALUES(1, "Hey There", "11.23.2016")
""")
# commit your work to database
connection.commit()

######## ITERATE THROUGH RESULTS  ########
cursor.execute("SELECT TOP 10 * FROM posts ORDER BY publish_date DESC")
for row in cursor:
    print("Message: " + row[1] + " | " + "Date: " + row[2])
    # if you pass as_dict=True to cursor
    # print(row["message"])

connection.close()
```

Vous pouvez faire n'importe quoi si votre travail est lié aux expressions SQL, transmettez simplement ces expressions à la méthode execute (opérations CRUD).

Pour avec instruction, appelant la procédure stockée, la gestion des erreurs ou d'autres exemples, vérifiez: pymssql.org

Lire Connexion de Python à SQL Server en ligne:
https://riptutorial.com/fr/python/topic/7985/connexion-de-python-a-sql-server

Chapitre 33: Connexion sécurisée au shell en Python

Paramètres

Paramètre	Usage
nom d'hôte	Ce paramètre indique à l'hôte auquel la connexion doit être établie
Nom d'utilisateur	nom d'utilisateur requis pour accéder à l'hôte
Port	port hôte
mot de passe	mot de passe pour le compte

Examples

connexion ssh

```
from paramiko import client
ssh = client.SSHClient() # create a new SSHClient object
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) #auto-accept unknown host keys
ssh.connect(hostname, username=username, port=port, password=password) #connect with a host
stdin, stdout, stderr = ssh.exec_command(command) # submit a command to ssh
print stdout.channel.recv_exit_status() #tells the status  1 - job failed
```

Lire Connexion sécurisée au shell en Python en ligne:
https://riptutorial.com/fr/python/topic/5709/connexion-securisee-au-shell-en-python

Chapitre 34: Copier des données

Examples

Effectuer une copie superficielle

Une copie superficielle est une copie d'une collection sans effectuer une copie de ses éléments.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

Effectuer une copie en profondeur

Si vous avez des listes imbriquées, il est souhaitable de cloner également les listes imbriquées. Cette action est appelée copie profonde.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

Réaliser une copie superficielle d'une liste

Vous pouvez créer des copies superficielles de listes à l'aide de tranches.

```
>>> l1 = [1,2,3]
>>> l2 = l1[:]     # Perform the shallow copy.
>>> l2
[1,2,3]
>>> l1 is l2
False
```

Copier un dictionnaire

Un objet dictionnaire a la `copy` méthode. Il effectue une copie superficielle du dictionnaire.

```
>>> d1 = {1:[]}
>>> d2 = d1.copy()
>>> d1 is d2
False
>>> d1[1] is d2[1]
```

```
True
```

Copier un ensemble

Les ensembles ont également une méthode de `copy` . Vous pouvez utiliser cette méthode pour effectuer une copie superficielle.

```
>>> s1 = {()}
>>> s2 = s1.copy()
>>> s1 is s2
False
>>> s2.add(3)
>>> s1
{[]}
>>> s2
{3,[]}
```

Lire Copier des données en ligne: https://riptutorial.com/fr/python/topic/920/copier-des-donnees

Chapitre 35: Cours de base avec Python

Remarques

Curses est un module de gestion de terminal (ou d'affichage de caractères) de base de Python. Cela peut être utilisé pour créer des interfaces utilisateur ou des interfaces utilisateur basées sur un terminal.

Ceci est un port python d'une bibliothèque C plus populaire 'ncurses'

Examples

Exemple d'invocation de base

```
import curses
import traceback

try:
    # -- Initialize --
    stdscr = curses.initscr()    # initialize curses screen
    curses.noecho()              # turn off auto echoing of keypress on to screen
    curses.cbreak()              # enter break mode where pressing Enter key
                                 #  after keystroke is not required for it to register
    stdscr.keypad(1)             # enable special Key values such as curses.KEY_LEFT etc

    # -- Perform an action with Screen --
    stdscr.border(0)
    stdscr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    stdscr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = stdscr.getch()
        if ch == ord('q'):
            break

    # -- End of user code --

except:
    traceback.print_exc()     # print trace back log of the error

finally:
    # --- Cleanup on exit ---
    stdscr.keypad(0)
    curses.echo()
    curses.nocbreak()
    curses.endwin()
```

La fonction d'assistance wrapper ().

Bien que l'invocation de base ci-dessus soit assez facile, le paquet curses fournit la `wrapper(func, ...)` aide `wrapper(func, ...)` . L'exemple ci-dessous contient l'équivalent de ci-dessus:

```
main(scr, *args):
    # -- Perform an action with Screen --
    scr.border(0)
    scr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    scr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = scr.getch()
        if ch == ord('q'):

curses.wrapper(main)
```

Ici, wrapper initialisera les curses, créera `stdscr` , un WindowObject et transmettra à la fois stdscr et tout autre argument à `func` . Lorsque `func` revient, `wrapper` restaure le terminal avant la fin du programme.

Lire Cours de base avec Python en ligne: https://riptutorial.com/fr/python/topic/5851/cours-de-base-avec-python

Chapitre 36: Création d'un service Windows à l'aide de Python

Introduction

Les processus sans tête (sans interface utilisateur) sous Windows sont appelés services. Ils peuvent être contrôlés (démarrés, arrêtés, etc.) à l'aide de contrôles Windows standard tels que la console de commande, Powershell ou l'onglet Services du Gestionnaire des tâches. Un bon exemple pourrait être une application qui fournit des services réseau, tels qu'une application Web, ou une application de sauvegarde qui effectue diverses tâches d'archivage en arrière-plan. Il existe plusieurs manières de créer et d'installer une application Python en tant que service sous Windows.

Examples

Un script Python pouvant être exécuté en tant que service

Les modules utilisés dans cet exemple font partie de pywin32 (extensions Python for Windows). Selon la manière dont vous avez installé Python, vous devrez peut-être l'installer séparément.

```
import win32serviceutil
import win32service
import win32event
import servicemanager
import socket


class AppServerSvc (win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"

    def __init__(self,args):
        win32serviceutil.ServiceFramework.__init__(self,args)
        self.hWaitStop = win32event.CreateEvent(None,0,0,None)
        socket.setdefaulttimeout(60)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

    def SvcDoRun(self):
        servicemanager.LogMsg(servicemanager.EVENTLOG_INFORMATION_TYPE,
                              servicemanager.PYS_SERVICE_STARTED,
                              (self._svc_name_,''))
        self.main()

    def main(self):
        pass

if __name__ == '__main__':
```

```
    win32serviceutil.HandleCommandLine(AppServerSvc)
```

Ceci est juste un passe-partout. Votre code d'application, invoquant probablement un script séparé, irait dans la fonction main ().

Vous devrez également l'installer en tant que service. La meilleure solution pour le moment semble être d'utiliser Non-sucking Service Manager . Cela vous permet d'installer un service et fournit une interface graphique pour configurer la ligne de commande exécutée par le service. Pour Python, vous pouvez le faire, ce qui crée le service en une seule fois:

```
nssm install MyServiceName c:\python27\python.exe c:\temp\myscript.py
```

Où my_script.py est le script standard ci-dessus, modifié pour appeler votre script ou code d'application dans la fonction main (). Notez que le service n'exécute pas directement le script Python, il exécute l'interpréteur Python et lui transmet le script principal sur la ligne de commande.

Vous pouvez également utiliser les outils fournis dans le Kit de ressources Windows Server pour la version de votre système d'exploitation afin de créer le service.

Exécution d'une application Web Flask en tant que service

Ceci est une variante de l'exemple générique. Il vous suffit d'importer votre script d'application et d'appeler sa méthode `run()` dans la fonction `main()` du service. Dans ce cas, nous utilisons également le module de multitraitement en raison d'un problème d'accès à `WSGIRequestHandler` .

```
import win32serviceutil
import win32service
import win32event
import servicemanager
from multiprocessing import Process

from app import app


class Service(win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"
    _svc_description_ = "Tests Python service framework by receiving and echoing messages over
a named pipe"

    def __init__(self, *args):
        super().__init__(*args)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        self.process.terminate()
        self.ReportServiceStatus(win32service.SERVICE_STOPPED)

    def SvcDoRun(self):
        self.process = Process(target=self.main)
        self.process.start()
        self.process.run()

    def main(self):
```

```
        app.run()


if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(Service)
```

Adapté de http://stackoverflow.com/a/25130524/318488

Lire Création d'un service Windows à l'aide de Python en ligne:
https://riptutorial.com/fr/python/topic/9065/creation-d-un-service-windows-a-l-aide-de-python

Chapitre 37: Créer des paquets Python

Remarques

L' exemple de projet pypa contient un modèle complet et facilement modifiable, `setup.py` qui démontre un large éventail de fonctionnalités que setup-tools peut offrir.

Examples

introduction

Chaque paquet nécessite un fichier `setup.py` qui décrit le paquet.

Considérez la structure de répertoires suivante pour un package simple:

```
+-- package_name
|       |
|       +-- __init__.py
|
+-- setup.py
```

Le `__init__.py` ne contient que la ligne `def foo(): return 100` .

Le `setup.py` suivant définira le package:

```
from setuptools import setup


setup(
    name='package_name',                    # package name
    version='0.1',                          # version
    description='Package Description',      # short description
    url='http://example.com',               # package URL
    install_requires=[],                    # list of packages this package depends
                                            # on.
    packages=['package_name'],              # List of module names that installing
                                            # this package will provide.
)
```

virtualenv est idéal pour tester les installations de paquets sans modifier vos autres environnements Python:

```
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
$ python setup.py install
running install
...
Installed .../package_name-0.1-....egg
...
```

```
$ python
>>> import package_name
>>> package_name.foo()
100
```

Téléchargement vers PyPI

Une fois que votre `setup.py` est entièrement fonctionnel (voir Introduction), il est très facile de télécharger votre paquet vers PyPI .

Configurer un fichier .pypirc

Ce fichier stocke les identifiants et les mots de passe pour authentifier vos comptes. Il est généralement stocké dans votre répertoire personnel.

```
# .pypirc file

[distutils]
index-servers =
  pypi
  pypitest

[pypi]
repository=https://pypi.python.org/pypi
username=your_username
password=your_password

[pypitest]
repository=https://testpypi.python.org/pypi
username=your_username
password=your_password
```

Il est plus sûr d'utiliser la `twine` pour le téléchargement de paquets, alors assurez-vous qu'elle est installée.

```
$ pip install twine
```

S'inscrire et télécharger sur testpypi (facultatif)

Remarque : PyPI n'autorise pas le remplacement des paquets téléchargés . Il est donc prudent de tester d'abord votre déploiement sur un serveur de test dédié, par exemple, testpypi. Cette option sera discutée. Envisagez un système de gestion des versions pour votre package avant de le télécharger, comme le contrôle de version du calendrier ou le contrôle de version sémantique .

Connectez-vous ou créez un nouveau compte sur testpypi . L'inscription n'est requise que la première fois, bien que l'enregistrement plus d'une fois ne soit pas dangereux.

```
$ python setup.py register -r pypitest
```

Dans le répertoire racine de votre package:

```
$ twine upload dist/* -r pypitest
```

Votre paquet devrait maintenant être accessible via votre compte.

Essai

Créez un environnement virtuel de test. Essayez d' `pip install` votre paquet à partir de testpypi ou de PyPI.

```
# Using virtualenv
$ mkdir testenv
$ cd testenv
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
# Test from testpypi
(.virtualenv)  pip install --verbose --extra-index-url https://testpypi.python.org/pypi
package_name
...
# Or test from PyPI
(.virtualenv) $ pip install package_name
...

(.virtualenv) $ python
Python 3.5.1 (default, Jan 27 2016, 19:16:39)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import package_name
>>> package_name.foo()
100
```

En cas de succès, votre paquet est le moins importable. Vous pourriez également envisager de tester votre API avant votre téléchargement final sur PyPI. Si votre paquet a échoué pendant le test, ne vous inquiétez pas. Vous pouvez toujours résoudre le problème, re-télécharger sur testpypi et tester à nouveau.

Enregistrer et télécharger sur PyPI

Assurez-vous que la `twine` est installée:

```
$ pip install twine
```

Connectez-vous ou créez un nouveau compte sur PyPI .

```
$ python setup.py register -r pypi
```

```
$ twine upload dist/*
```

C'est tout! Votre colis est maintenant en ligne .

Si vous découvrez un bogue, téléchargez simplement une nouvelle version de votre paquet.

Documentation

N'oubliez pas d'inclure au moins une sorte de documentation pour votre paquet. PyPi prend comme langage de formatage par défaut reStructuredText .

Readme

Si votre paquet ne contient pas une grande documentation, incluez ce qui peut aider les autres utilisateurs dans le fichier `README.rst` . Lorsque le fichier est prêt, il en faut un autre pour que PyPi le montre.

Créez le fichier `setup.cfg` et insérez-y ces deux lignes:

```
[metadata]
description-file = README.rst
```

Notez que si vous essayez de placer le fichier Markdown dans votre paquet, PyPi le lira comme un fichier texte pur sans aucune mise en forme.

Licence

Il est souvent plus que bienvenu de placer un fichier `LICENSE.txt` dans votre package avec l'une des licences OpenSource pour indiquer aux utilisateurs s'ils peuvent utiliser votre package par exemple dans des projets commerciaux ou si votre code est utilisable avec leur licence.

De manière plus lisible, certaines licences sont expliquées chez TL; DR .

Rendre le package exécutable

Si votre paquet n'est pas seulement une bibliothèque, mais un morceau de code pouvant être utilisé soit comme une vitrine, soit comme une application autonome lorsque votre paquet est installé, placez ce morceau de code dans le fichier `__main__.py` .

Placez le `__main__.py` dans le dossier `package_name` . De cette façon, vous pourrez l'exécuter directement depuis la console:

```
python -m package_name
```

S'il n'y a pas de fichier `__main__.py` disponible, le paquet ne sera pas exécuté avec cette

commande et cette erreur sera imprimée:

> python: aucun module nommé `package_name.__main__`; 'nom_package' est un package et ne peut pas être directement exécuté.

Lire Créer des paquets Python en ligne: https://riptutorial.com/fr/python/topic/1381/creer-des-paquets-python

Chapitre 38: Créer un environnement virtuel avec virtualenvwrapper dans Windows

Examples

Environnement virtuel avec virtualenvwrapper pour windows

Supposons que vous ayez besoin de travailler sur trois projets différents: le projet A, le projet B et le projet C. le projet A et le projet B nécessitent python 3 et certaines bibliothèques requises. Mais pour le projet C, vous avez besoin de python 2.7 et de bibliothèques dépendantes.

La meilleure pratique consiste donc à séparer ces environnements de projet. Pour créer un environnement virtuel Python distinct, suivez les étapes ci-dessous:

Étape 1: Installez pip avec cette commande: `python -m pip install -U pip`

Etape 2: Installez ensuite le package "virtualenvwrapper-win" en utilisant la commande (la commande peut être exécutée sous Windows Power Shell):

```
pip install virtualenvwrapper-win
```

Étape 3: Créez un nouvel environnement virtualenv en utilisant la commande: `mkvirtualenv python_3.5`

Étape 4: Activez l'environnement à l'aide de la commande:

```
workon < environment name>
```

Commandes principales pour virtualenvwrapper:

```
mkvirtualenv <name>
Create a new virtualenv environment named <name>. The environment will be created in
WORKON_HOME.

lsvirtualenv
List all of the enviornments stored in WORKON_HOME.

rmvirtualenv <name>
Remove the environment <name>. Uses folder_delete.bat.

workon [<name>]
If <name> is specified, activate the environment named <name> (change the working virtualenv
to <name>). If a project directory has been defined, we will change into it. If no argument is
specified, list the available environments. One can pass additional option -c after virtualenv
name to cd to virtualenv directory if no projectdir is set.

deactivate
Deactivate the working virtualenv and switch back to the default system Python.

add2virtualenv <full or relative path>
```

```
If a virtualenv environment is active, appends <path> to virtualenv_path_extensions.pth inside
the environment's site-packages, which effectively adds <path> to the environment's
PYTHONPATH. If a virtualenv environment is not active, appends <path> to
virtualenv_path_extensions.pth inside the default Python's site-packages. If <path> doesn't
exist, it will be created.
```

Lire Créer un environnement virtuel avec virtualenvwrapper dans Windows en ligne: https://riptutorial.com/fr/python/topic/9984/creer-un-environnement-virtuel-avec-virtualenvwrapper-dans-windows

Chapitre 39: ctypes

Introduction

`ctypes` est une bibliothèque intégrée python qui appelle les fonctions exportées à partir de bibliothèques compilées natives.

Remarque: Comme cette bibliothèque gère le code compilé, il est relativement dépendant du système d'exploitation.

Examples

Utilisation de base

Disons que nous voulons utiliser la fonction `ntohl` `libc` .

Tout d'abord, nous devons charger `libc.so` :

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

Ensuite, nous obtenons l'objet function:

```
>>> ntohl = libc.ntohl
>>> ntohl
<_FuncPtr object at 0xbaadf00d>
```

Et maintenant, nous pouvons simplement invoquer la fonction:

```
>>> ntohl(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

Ce qui fait exactement ce que nous attendons de lui.

Pièges communs

Ne pas charger un fichier

La première erreur possible échoue lors du chargement de la bibliothèque. Dans ce cas, une erreur OSE est généralement déclenchée.

C'est soit parce que le fichier n'existe pas (ou ne peut pas être trouvé par le système

d'exploitation):

```
>>> cdll.LoadLibrary("foobar.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: foobar.so: cannot open shared object file: No such file or directory
```

Comme vous pouvez le voir, l'erreur est claire et assez indicative.

La deuxième raison est que le fichier est trouvé mais n'a pas le bon format.

```
>>> cdll.LoadLibrary("libc.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: /usr/lib/i386-linux-gnu/libc.so: invalid ELF header
```

Dans ce cas, le fichier est un fichier script et non un fichier `.so` . Cela peut également se produire lorsque vous essayez d'ouvrir un fichier `.dll` sur une machine Linux ou un fichier 64 bits sur un interpréteur Python 32 bits. Comme vous pouvez le voir, dans ce cas, l'erreur est un peu plus vague et nécessite des fouilles.

Ne pas accéder à une fonction

En supposant que nous avons chargé avec succès le fichier `.so` , nous devons ensuite accéder à notre fonction comme nous l'avons fait sur le premier exemple.

Lorsqu'une fonction non existante est utilisée, un `AttributeError` est déclenché:

```
>>> libc.foo
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 360, in __getattr__
    func = self.__getitem__(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 365, in __getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: /lib/i386-linux-gnu/libc.so.6: undefined symbol: foo
```

Objet ctypes de base

L'objet le plus fondamental est un int:

```
>>> obj = ctypes.c_int(12)
>>> obj
```

```
c_long(12)
```

Maintenant, `obj` réfère à un morceau de mémoire contenant la valeur 12.

Cette valeur est accessible directement et même modifiée:

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

Depuis `obj` réfère à une partie de la mémoire, nous pouvons également trouver sa taille et son emplacement:

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

tableaux de type ctypes

Comme tout bon programmeur C le sait, une seule valeur ne vous mènera pas aussi loin. Ce qui va vraiment nous faire avancer, ce sont les tableaux!

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

Ce n'est pas un tableau réel, mais c'est assez proche! Nous avons créé une classe qui dénote un tableau de 16 `int` s.

Il ne reste plus qu'à l'initialiser:

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcafe>
```

Maintenant, `arr` est un tableau qui contient les nombres de 0 à 15.

Ils sont accessibles comme n'importe quelle liste:

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

Et tout comme n'importe quel autre objet `ctypes` , il a également une taille et un emplacement:

```
>>> sizeof(arr)
```

```
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc000l0ff'
```

Fonctions d'emballage pour les types

Dans certains cas, une fonction C accepte un pointeur de fonction. En tant `ctypes` avides de `ctypes` , nous aimerions utiliser ces fonctions, et même passer la fonction python en argument.

Définissons une fonction:

```
>>> def max(x, y):
        return x if x >= y else y
```

Maintenant, cette fonction prend deux arguments et renvoie un résultat du même type. Pour l'exemple, supposons que ce type est un int.

Comme nous l'avons fait sur l'exemple du tableau, nous pouvons définir un objet qui dénote ce prototype:

```
>>> CFUNCTYPE(c_int, c_int, c_int)
<CFunctionType object at 0xdeadbeef>
```

Ce prototype dénote une fonction qui retourne un `c_int` (le premier argument) et accepte deux arguments `c_int` (les autres arguments).

Maintenant, enveloppons la fonction:

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

Les prototypes de fonctions ont plus d'usage: ils peuvent envelopper la fonction `ctypes` (comme `libc.ntohl`) et vérifier que les arguments corrects sont utilisés lors de l'appel de la fonction.

```
>>> libc.ntohl() # garbage in - garbage out
>>> CFUNCTYPE(c_int, c_int)(libc.ntohl)()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: this function takes at least 1 argument (0 given)
```

Utilisation complexe

`lfind` tous les exemples ci-dessus en un scénario complexe: utiliser la fonction `lfind` `libc` .

Pour plus de détails sur la fonction, lisez la page de manuel . Je vous exhorte à le lire avant de continuer.

Tout d'abord, nous définirons les prototypes appropriés:

```
>>> compar_proto = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> lfind_proto = CFUNCTYPE(c_void_p, c_void_p, c_void_p, POINTER(c_uint), c_uint,
compar_proto)
```

Ensuite, créons les variables:

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

Et maintenant, nous définissons la fonction de comparaison:

```
>>> def compar(x, y):
        return x.contents.value - y.contents.value
```

Notez que `x` et `y` sont `POINTER(c_int)` , nous devons donc les déréférencer et prendre leurs valeurs afin de comparer la valeur stockée dans la mémoire.

Maintenant, nous pouvons tout combiner:

```
>>> lfind = lfind_proto(libc.lfind)
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```

`ptr` est le pointeur vide retourné. Si la `key` n'a pas été trouvée dans `arr` , la valeur serait `None` , mais dans ce cas, nous avons une valeur valide.

Maintenant, nous pouvons le convertir et accéder à la valeur:

```
>>> cast(ptr, POINTER(c_int)).contents
c_long(12)
```

En outre, nous pouvons voir que `ptr` pointe vers la valeur correcte à l'intérieur d' `arr` :

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr
True
```

Lire ctypes en ligne: https://riptutorial.com/fr/python/topic/9050/ctypes

Chapitre 40: Date et l'heure

Remarques

Python fournit à la fois des méthodes intégrées et des bibliothèques externes pour créer, modifier, analyser et manipuler les dates et les heures.

Examples

Analyse d'une chaîne en objet datetime sensible au fuseau horaire

Python 3.2+ prend en charge le format `%z` lors du traitement d'une chaîne dans un objet `datetime` .

> Offset UTC sous la forme `+HHMM` ou `-HHMM` (chaîne vide si l'objet est naïf).

Python 3.x 3.2

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

Pour les autres versions de Python, vous pouvez utiliser une bibliothèque externe telle que `dateutil` , ce qui `dateutil` analyse syntaxique d'une chaîne avec un fuseau horaire dans un objet `datetime` .

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

La variable `dt` est maintenant un objet `datetime` avec la valeur suivante:

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

Arithmétique de date simple

Les dates n'existent pas isolément. Il est fréquent que vous deviez trouver la durée entre les dates ou déterminer quelle sera la date de demain. Cela peut être accompli en utilisant des objets `timedelta`

```
import datetime

today = datetime.date.today()
print('Today:', today)

yesterday = today - datetime.timedelta(days=1)
print('Yesterday:', yesterday)

tomorrow = today + datetime.timedelta(days=1)
print('Tomorrow:', tomorrow)
```

```
print('Time between tomorrow and yesterday:', tomorrow - yesterday)
```

Cela produira des résultats similaires à:

```
Today: 2016-04-15
Yesterday: 2016-04-14
Tomorrow: 2016-04-16
Difference between tomorrow and yesterday: 2 days, 0:00:00
```

Utilisation d'objets de base datetime

Le module datetime contient trois principaux types d'objets - date, heure et date-heure.

```
import datetime

# Date object
today = datetime.date.today()
new_year = datetime.date(2017, 01, 01) #datetime.date(2017, 1, 1)

# Time object
noon = datetime.time(12, 0, 0) #datetime.time(12, 0)

# Current datetime
now = datetime.datetime.now()

# Datetime object
millenium_turn = datetime.datetime(2000, 1, 1, 0, 0, 0) #datetime.datetime(2000, 1, 1, 0, 0)
```

Les opérations arithmétiques pour ces objets sont uniquement prises en charge dans le même type de données et l'exécution d'une arithmétique simple avec des instances de types différents entraînera une erreur TypeError.

```
# subtraction of noon from today
noon-today
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.date'
However, it is straightforward to convert between types.

# Do this instead
print('Time since the millenium at midnight: ',
      datetime.datetime(today.year, today.month, today.day) - millenium_turn)

# Or this
print('Time since the millenium at noon: ',
      datetime.datetime.combine(today, noon) - millenium_turn)
```

Itérer sur les dates

Parfois, vous souhaitez effectuer une itération sur une plage de dates allant d'une date de début à une date de fin. Vous pouvez le faire en utilisant la bibliothèque `datetime` et l'objet `timedelta` :

```
import datetime

# The size of each step in days
day_delta = datetime.timedelta(days=1)

start_date = datetime.date.today()
end_date = start_date + 7*day_delta

for i in range((end_date - start_date).days):
    print(start_date + i*day_delta)
```

Qui produit:

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

Analyse d'une chaîne avec un nom de fuseau horaire court en un objet datetime sensible au fuseau horaire

En utilisant la bibliothèque `dateutil` comme dans l' exemple précédent sur l'analyse des horodatages sensibles au fuseau horaire , il est également possible d'analyser les horodatages avec un nom de fuseau horaire spécifié "court".

Pour les dates formatées avec des noms de fuseau horaire courte ou des abréviations, qui sont généralement ambigus (par exemple CST, qui pourrait être Central Standard Time, Heure normale de Chine, Cuba heure normale, etc - plus peuvent être trouvés ici) ou pas nécessairement disponibles dans une base de données standard , il est nécessaire de spécifier un mappage entre l'abréviation du fuseau horaire et l'objet `tzinfo` .

```
from dateutil import tz
from dateutil.parser import parse

ET = tz.gettz('US/Eastern')
CT = tz.gettz('US/Central')
MT = tz.gettz('US/Mountain')
PT = tz.gettz('US/Pacific')

us_tzinfos = {'CST': CT, 'CDT': CT,
              'EST': ET, 'EDT': ET,
              'MST': MT, 'MDT': MT,
              'PST': PT, 'PDT': PT}

dt_est = parse('2014-01-02 04:00:00 EST', tzinfos=us_tzinfos)
dt_pst = parse('2016-03-11 16:00:00 PST', tzinfos=us_tzinfos)
```

Après avoir exécuté ceci:

```
dt_est
# datetime.datetime(2014, 1, 2, 4, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Eastern'))
```

```
dt_pst
# datetime.datetime(2016, 3, 11, 16, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Pacific'))
```

Il est à noter que si vous utilisez un `pytz` horaire `pytz` avec cette méthode, celle-ci *ne* sera *pas* correctement localisée:

```
from dateutil.parser import parse
import pytz

EST = pytz.timezone('America/New_York')
dt = parse('2014-02-03 09:17:00 EST', tzinfos={'EST': EST})
```

Cela attache simplement le `pytz` horaire de `pytz` à la date et heure:

```
dt.tzinfo # Will be in Local Mean Time!
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Si vous utilisez cette méthode, vous devriez probablement re `localize` la partie naïve du datetime après l' analyse syntaxique:

```
dt_fixed = dt.tzinfo.localize(dt.replace(tzinfo=None))
dt_fixed.tzinfo # Now it's EST.
# <DstTzInfo 'America/New_York' EST-1 day, 19:00:00 STD>)
```

Construire des datetimes dans le fuseau horaire

Par défaut, tous les objets `datetime` sont naïfs. Pour les rendre compatibles avec le fuseau horaire, vous devez joindre un objet `tzinfo` , qui fournit le décalage UTC et l'abréviation du fuseau horaire en fonction de la date et de l'heure.

Zones de décalage fixe

Pour les fuseaux horaires à décalage fixe par rapport à UTC, dans Python 3.2+, le module `datetime` fournit la classe `timezone` , une implémentation concrète de `tzinfo` , qui prend un `timedelta` et un paramètre (facultatif) name:

Python 3.x 3.2

```
from datetime import datetime, timedelta, timezone
JST = timezone(timedelta(hours=+9))

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00

print(dt.tzname())
# UTC+09:00

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=timezone(timedelta(hours=9), 'JST'))
print(dt.tzname)
# 'JST'
```

Pour les versions Python antérieures à 3.2, il est nécessaire d'utiliser une bibliothèque tierce, telle que `dateutil` . `dateutil` fournit une classe équivalente, `tzoffset` , qui (à partir de la version 2.5.3) prend les arguments de la forme `dateutil.tz.tzoffset(tzname, offset)` , où le `offset` est spécifié en secondes:

Python 3.x 3.2
Python 2.x 2.7

```
from datetime import datetime, timedelta
from dateutil import tz

JST = tz.tzoffset('JST', 9 * 3600) # 3600 seconds per hour
dt = datetime(2015, 1, 1, 12, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00
print(dt.tzname)
# 'JST'
```

Zones avec heure avancée

Pour les zones avec heure avancée, les bibliothèques standard python ne fournissent pas de classe standard, il est donc nécessaire d'utiliser une bibliothèque tierce. `pytz` et `dateutil` sont des bibliothèques populaires fournissant des classes de fuseau horaire.

En plus des fuseaux horaires statiques, `dateutil` fournit des classes de fuseau horaire qui utilisent l'heure d'été (voir la documentation du module `tz`). Vous pouvez utiliser la méthode `tz.gettz()` pour obtenir un objet de fuseau horaire, qui peut ensuite être transmis directement au constructeur `datetime` :

```
from datetime import datetime
from dateutil import tz
local = tz.gettz() # Local time
PT = tz.gettz('US/Pacific') # Pacific time

dt_l = datetime(2015, 1, 1, 12, tzinfo=local) # I am in EST
dt_pst = datetime(2015, 1, 1, 12, tzinfo=PT)
dt_pdt = datetime(2015, 7, 1, 12, tzinfo=PT) # DST is handled automatically
print(dt_l)
# 2015-01-01 12:00:00-05:00
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-07-01 12:00:00-07:00
```

ATTENTION : à partir de la version 2.5.3, `dateutil` ne gère pas correctement les `dateutil` ambiguës et utilisera toujours la date *ultérieure* par défaut. Il n'y a aucun moyen de construire un objet avec un `dateutil` fuseau horaire représentant, par exemple `2015-11-01 1:30 EDT-4` , puisque c'est au *cours* d' une transition heure d'été.

Tous les cas de bord sont traités correctement lors de l'utilisation de `pytz` , mais les `pytz` horaires `pytz` *ne* doivent *pas* être directement liés aux fuseaux horaires via le constructeur. Au lieu de cela, un `pytz` horaire `pytz` doit être attaché en utilisant la méthode de `localize` du fuseau horaire:

```
from datetime import datetime, timedelta
import pytz

PT = pytz.timezone('US/Pacific')
dt_pst = PT.localize(datetime(2015, 1, 1, 12))
dt_pdt = PT.localize(datetime(2015, 11, 1, 0, 30))
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-11-01 00:30:00-07:00
```

Sachez que si vous effectuez une arithmétique datetime sur un `pytz` horaire `pytz` pytz, vous devez soit effectuer les calculs en UTC (si vous voulez un temps écoulé absolu), soit appeler `normalize()` sur le résultat:

```
dt_new = dt_pdt + timedelta(hours=3) # This should be 2:30 AM PST
print(dt_new)
# 2015-11-01 03:30:00-07:00
dt_corrected = PT.normalize(dt_new)
print(dt_corrected)
# 2015-11-01 02:30:00-08:00
```

Analyse floue de datetime (extraction de datetime d'un texte)

Il est possible d'extraire une date d'un texte à l' aide du `dateutil` analyseur en mode « flou », où les composants de la chaîne non reconnu comme faisant partie d'une date sont ignorées.

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

`dt` est maintenant un *objet* `datetime` et vous `datetime.datetime(2047, 1, 1, 8, 21)` imprimé.

Changer de fuseau horaire

Pour basculer entre les fuseaux horaires, vous avez besoin d'objets datetime qui prennent en compte le fuseau horaire.

```
from datetime import datetime
from dateutil import tz

utc = tz.tzutc()
local = tz.tzlocal()

utc_now = datetime.utcnow()
utc_now # Not timezone-aware.

utc_now = utc_now.replace(tzinfo=utc)
utc_now # Timezone-aware.

local_now = utc_now.astimezone(local)
local_now # Converted to local time.
```

Analyse d'un horodatage ISO 8601 arbitraire avec des bibliothèques minimales

Python ne prend en charge que l'analyse des horodatages ISO 8601. Pour le `strptime` vous devez savoir exactement dans quel format il se trouve. Comme complication, la cordification d'un `datetime` est un horodatage ISO 8601, avec un espace comme séparateur et fraction de 6 chiffres:

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))
# '2016-07-22 09:25:59.555555'
```

mais si la fraction est 0, aucune partie fractionnaire n'est sortie

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

Mais ces deux formes nécessitent un format *différent* pour le `strptime` . De plus, `strptime' does not support at all parsing minute timezones that have a : in it, thus` 2016-07-22 09: 25: 59 + 0300 `can be parsed, but the standard format` 2016-07-22 09:25:59 +03: 00` ne peut pas.

Il existe une bibliothèque de fichiers `iso8601` appelée `iso8601` qui analyse correctement les horodatages ISO 8601 et uniquement ceux-ci.

Il supporte les fractions et les fuseaux horaires, et le séparateur `T` avec une seule fonction:

```
import iso8601
iso8601.parse_date('2016-07-22 09:25:59')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22 09:25:59+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<FixedOffset '+03:00' ...>)
iso8601.parse_date('2016-07-22 09:25:59Z')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22T09:25:59.000111+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, 111, tzinfo=<FixedOffset '+03:00' ...>)
```

Si aucun fuseau horaire n'est défini, `iso8601.parse_date` est défini `iso8601.parse_date` défaut sur UTC. La zone par défaut peut être modifiée avec l'argument mot-clé `default_zone` . Notamment, s'il s'agit de `None` au lieu de la valeur par défaut, les horodatages qui n'ont pas de fuseau horaire explicite sont renvoyés en tant que dat date naïves à la place:

```
iso8601.parse_date('2016-07-22T09:25:59', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59)
iso8601.parse_date('2016-07-22T09:25:59Z', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
```

Conversion de l'horodatage en datetime

Le module `datetime` peut convertir un `timestamp` POSIX en objet `datetime` ITC.

L'époque est le 1er janvier 1970 à minuit.

```
import time
from datetime import datetime
seconds_since_epoch=time.time()  #1469182681.709

utc_date=datetime.utcfromtimestamp(seconds_since_epoch) #datetime.datetime(2016, 7, 22, 10,
18, 1, 709000)
```

Soustraire les mois d'une date avec précision

Utiliser le module `calendar`

```
import calendar
from datetime import date

def monthdelta(date, delta):
    m, y = (date.month+delta) % 12, date.year + ((date.month)+delta-1) // 12
    if not m: m = 12
    d = min(date.day, calendar.monthrange(y, m)[1])
    return date.replace(day=d,month=m, year=y)

next_month = monthdelta(date.today(), 1) #datetime.date(2016, 10, 23)
```

Utilisation du module `dateutils`

```
import datetime
import dateutil.relativedelta

d = datetime.datetime.strptime("2013-03-31", "%Y-%m-%d")
d2 = d - dateutil.relativedelta.relativedelta(months=1)  #datetime.datetime(2013, 2, 28, 0, 0)
```

Différences de temps de calcul

`timedelta` module `timedelta` est pratique pour calculer les différences entre les temps:

```
from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23)     # datetime.datetime(2016, 05, 23, 0, 0, 0)
```

La spécification de l'heure est facultative lors de la création d'un nouvel objet `datetime`

```
delta = now-then
```

`delta` est de type `timedelta`

```
print(delta.days)
# 60
print(delta.seconds)
# 40826
```

Pour obtenir un n et un jour d'avance, nous pouvons utiliser:

n jour après jour:

```
def get_n_days_after_date(date_format="%d %B %Y", add_days=120):

    date_n_days_after = datetime.datetime.now() + timedelta(days=add_days)
    return date_n_days_after.strftime(date_format)
```

n jour avant la date:

```
 def get_n_days_before_date(self, date_format="%d %B %Y", days_before=120):

        date_n_days_ago = datetime.datetime.now() - timedelta(days=days_before)
        return date_n_days_ago.strftime(date_format)
```

Obtenir un horodatage ISO 8601

Sans fuseau horaire, avec microsecondes

```
from datetime import datetime

datetime.now().isoformat()
# Out: '2016-07-31T23:08:20.886783'
```

Avec fuseau horaire, avec microsecondes

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# Out: '2016-07-31T23:09:43.535074-07:00'
```

Avec fuseau horaire, sans microsecondes

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Out: '2016-07-31T23:10:30-07:00'
```

Voir ISO 8601 pour plus d'informations sur le format ISO 8601.

Lire Date et l'heure en ligne: https://riptutorial.com/fr/python/topic/484/date-et-l-heure

Chapitre 41: Décorateurs

Introduction

Les fonctions de décorateur sont des modèles de conception de logiciel. Ils modifient dynamiquement les fonctionnalités d'une fonction, d'une méthode ou d'une classe sans avoir à utiliser directement les sous-classes ou à modifier le code source de la fonction décorée. Utilisés correctement, les décorateurs peuvent devenir des outils puissants dans le processus de développement. Cette rubrique couvre l'implémentation et les applications des fonctions de décorateur dans Python.

Syntaxe

- def decorator_function (f): pass # définit un décorateur nommé decorator_function
- @decorator_function
  def decorated_function (): pass # la fonction est maintenant encapsulée (décorée par)
- decorated_function = decorator_function (decorated_function) # équivaut à utiliser le sucre syntaxique `@decorator_function`

Paramètres

Paramètre	Détails
F	La fonction à décorer

Examples

Fonction de décorateur

Les décorateurs augmentent le comportement d'autres fonctions ou méthodes. Toute fonction prenant une fonction en paramètre et renvoyant une fonction augmentée peut être utilisée comme **décorateur** .

```
# This simplest decorator does nothing to the function being decorated. Such
# minimal decorators can occasionally be used as a kind of code markers.
def super_secret_function(f):
    return f

@super_secret_function
def my_function():
    print("This is my secret function.")
```

La `@` -notation est un sucre syntaxique équivalent à ce qui suit:

```
my_function = super_secret_function(my_function)
```

Il est important de garder cela à l'esprit pour comprendre le fonctionnement des décorateurs. Cette syntaxe "unsugared" indique clairement pourquoi la fonction de décorateur prend une fonction comme argument et pourquoi elle doit renvoyer une autre fonction. Cela montre également ce qui arriverait si vous *ne* retournez *pas* une fonction:

```
def disabled(f):
    """
    This function returns nothing, and hence removes the decorated function
    from the local scope.
    """
    pass

@disabled
def my_function():
    print("This function can no longer be called...")

my_function()
# TypeError: 'NoneType' object is not callable
```

Ainsi, nous définissons généralement une *nouvelle fonction* dans le décorateur et la retournons. Cette nouvelle fonction ferait d'abord quelque chose qu'il doit faire, puis appelle la fonction d'origine et traite enfin la valeur de retour. Considérons cette fonction de décorateur simple qui imprime les arguments reçus par la fonction d'origine, puis les appelle.

```
#This is the decorator
def print_args(func):
    def inner_func(*args, **kwargs):
        print(args)
        print(kwargs)
        return func(*args, **kwargs) #Call the original function with its arguments.
    return inner_func

@print_args
def multiply(num_a, num_b):
    return num_a * num_b

print(multiply(3, 5))
#Output:
# (3,5) - This is actually the 'args' that the function receives.
# {} - This is the 'kwargs', empty because we didn't specify keyword arguments.
# 15 - The result of the function.
```

Classe de décorateur

Comme mentionné dans l'introduction, un décorateur est une fonction qui peut être appliquée à une autre fonction pour augmenter son comportement. Le sucre syntaxique est équivalent à ce qui suit: `my_func = decorator(my_func)` . Mais si le `decorator` était plutôt une classe? La syntaxe fonctionne toujours, sauf que maintenant `my_func` est remplacé par une instance de la classe de `decorator` . Si cette classe implémente la méthode magique `__call__()` , alors il serait toujours possible d'utiliser `my_func` comme si c'était une fonction:

```
class Decorator(object):
    """Simple decorator class."""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Before the function call.')
        res = self.func(*args, **kwargs)
        print('After the function call.')
        return res

@Decorator
def testfunc():
    print('Inside the function.')

testfunc()
# Before the function call.
# Inside the function.
# After the function call.
```

Notez qu'une fonction décorée avec un décorateur de classe ne sera plus considérée comme une "fonction" du point de vue de la vérification de type:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Méthodes de décoration

Pour les méthodes de décoration, vous devez définir une `__get__` __get__ supplémentaire:

```
from types import MethodType

class Decorator(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Inside the decorator.')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        # Return a Method if it is called on an instance
        return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass

a = Test()
```

À l'intérieur du décorateur.

Attention!

Les décorateurs de classes ne produisent qu'une seule instance pour une fonction spécifique. Ainsi, décorer une méthode avec un décorateur de classes partagera le même décorateur entre toutes les instances de cette classe:

```
from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0    # Number of calls of this method

    def __call__(self, *args, **kwargs):
        self.ncalls += 1   # Increment the calls counter
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class Test(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return 'something was done'

a = Test()
a.do_something()
a.do_something.ncalls   # 1
b = Test()
b.do_something()
b.do_something.ncalls   # 2
```

Faire ressembler un décorateur à la fonction décorée

Les décorateurs éliminent normalement les métadonnées des fonctions car elles ne sont pas identiques. Cela peut entraîner des problèmes lors de l'utilisation de la méta-programmation pour accéder de manière dynamique aux métadonnées de la fonction. Les métadonnées incluent également les docstrings de la fonction et son nom. `functools.wraps` permet à la fonction décorée de ressembler à la fonction d'origine en copiant plusieurs attributs dans la fonction wrapper.

```
from functools import wraps
```

Les deux méthodes d'emballage d'un décorateur permettent d'obtenir la même chose en masquant la fonction d'origine. Il n'y a aucune raison de préférer la version de la fonction à la version de la classe, sauf si vous l'utilisez déjà.

En tant que fonction

```
def decorator(func):
    # Copies the docstring, name, annotations and module to the decorator
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func

@decorator
def test():
    pass

test.__name__
```

'tester'

En tant que classe

```
class Decorator(object):
    def __init__(self, func):
        # Copies name, module, annotations and docstring to the instance.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
```

'Docstring de test.'

Décorateur avec des arguments (usine de décorateur)

Un décorateur ne prend qu'un argument: la fonction à décorer. Il n'y a aucun moyen de passer d'autres arguments.

Mais des arguments supplémentaires sont souvent souhaités. L'astuce consiste alors à créer une fonction qui prend des arguments arbitraires et renvoie un décorateur.

Fonctions de décorateur

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
```

```
            print('The decorator wants to tell you: {}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@decoratorfactory('Hello World')
def test():
    pass

test()
```

Le décorateur veut vous dire: Hello World

Note importante:

Avec de telles usines de décoration, vous **devez** appeler le décorateur avec une paire de parenthèses:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

TypeError: decorator () manquant 1 argument positionnel requis: 'func'

Cours de décorateur

```
def decoratorfactory(*decorator_args, **decorator_kwargs):

    class Decorator(object):
        def __init__(self, func):
            self.func = func

        def __call__(self, *args, **kwargs):
            print('Inside the decorator with arguments {}'.format(decorator_args))
            return self.func(*args, **kwargs)

    return Decorator

@decoratorfactory(10)
def test():
    pass

test()
```

À l'intérieur du décorateur avec des arguments (10,)

Créer une classe singleton avec un décorateur

Un singleton est un modèle qui limite l'instanciation d'une classe à une instance / objet. En

utilisant un décorateur, nous pouvons définir une classe en tant que singleton en forçant la classe à renvoyer une instance existante de la classe ou à créer une nouvelle instance (si elle n'existe pas).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

Ce décorateur peut être ajouté à toute déclaration de classe et s'assurera qu'au plus une instance de la classe soit créée. Tous les appels suivants renverront l'instance de classe déjà existante.

```
@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Created!")

instance = SomeSingletonClass()  # prints: Created!
instance = SomeSingletonClass()  # doesn't print anything
print(instance.x)                # 2

instance.x = 3
print(SomeSingletonClass().x)    # 3
```

Peu importe que vous vous référiez à l'instance de classe via votre variable locale ou que vous créiez une autre "instance", vous obtenez toujours le même objet.

Utiliser un décorateur pour chronométrer une fonction

```
import time
def timer(func):
    def inner(*args, **kwargs):
        t1 = time.time()
        f = func(*args, **kwargs)
        t2 = time.time()
        print 'Runtime took {0} seconds'.format(t2-t1)
        return f
    return inner

@timer
def example_function():
    #do stuff


example_function()
```

Lire Décorateurs en ligne: https://riptutorial.com/fr/python/topic/229/decorateurs

Chapitre 42: Définition de fonctions avec des arguments de liste

Examples

Fonction et appel

Les listes comme arguments ne sont qu'une autre variable:

```
def func(myList):
    for item in myList:
        print(item)
```

et peut être passé dans l'appel de fonction lui-même:

```
func([1,2,3,5,7])

1
2
3
5
7
```

Ou comme variable:

```
aList = ['a','b','c','d']
func(aList)

a
b
c
d
```

Lire Définition de fonctions avec des arguments de liste en ligne:
https://riptutorial.com/fr/python/topic/7744/definition-de-fonctions-avec-des-arguments-de-liste

Chapitre 43: Déploiement

Examples

Téléchargement d'un package Conda

Avant de commencer, vous devez avoir:

Anaconda installé sur votre système Compte sur Binstar Si vous n'utilisez pas Anaconda 1.6+, installez le client de ligne de commande binstar :

```
$ conda install binstar
$ conda update binstar
```

Si vous n'utilisez pas Anaconda, le Binstar est également disponible sur pypi:

```
$ pip install binstar
```

Maintenant, nous pouvons nous connecter:

```
$ binstar login
```

Testez votre login avec la commande whoami:

```
$ binstar whoami
```

Nous allons télécharger un paquet avec une simple fonction "hello world". Pour suivre, commencez par obtenir mon dépôt de démonstration à partir de Github:

```
$ git clone https://github.com/<NAME>/<Package>
```

C'est un petit répertoire qui ressemble à ceci:

```
package/
        setup.py
        test_package/
           __init__.py
           hello.py
           bld.bat
           build.sh
           meta.yaml
```

`Setup.py` est le fichier de génération de python standard et `hello.py` a notre fonction hello_world () unique.

Les `bld.bat` , `build.sh` et `meta.yaml` sont des scripts et des métadonnées pour le package `Conda` . Vous pouvez lire la page de construction de Conda pour plus d'informations sur ces trois fichiers

et leur objectif.

Maintenant, nous créons le paquet en exécutant:

```
$ conda build test_package/
```

C'est tout ce qu'il faut pour créer un package Conda.

L'étape finale consiste à télécharger sur binstar en copiant et en collant la dernière ligne de l'impression après avoir exécuté la compilation conda test_package / command. Sur mon système, la commande est la suivante:

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

Comme c'est la première fois que vous créez un package et que vous le lancez, vous serez invité à remplir des champs de texte qui pourraient être utilisés via l'application Web.

Vous verrez un `done` imprimer pour confirmer que vous avez téléchargé avec succès votre package à Conda BINSTAR.

Lire Déploiement en ligne: https://riptutorial.com/fr/python/topic/4064/deploiement

Chapitre 44: Dérogation de méthode

Examples

Méthode de base

Voici un exemple de substitution de base en Python (pour des raisons de clarté et de compatibilité avec Python 2 et 3, en utilisant une nouvelle classe de style et en `print` avec `()`):

```
class Parent(object):
    def introduce(self):
        print("Hello!")

    def print_name(self):
        print("Parent")


class Child(Parent):
    def print_name(self):
        print("Child")


p = Parent()
c = Child()

p.introduce()
p.print_name()

c.introduce()
c.print_name()

$ python basic_override.py
Hello!
Parent
Hello!
Child
```

Lorsque la classe `Child` est créée, elle hérite des méthodes de la classe `Parent` . Cela signifie que toutes les méthodes de la classe parente auront la classe enfant. Dans l'exemple, le `introduce` est défini pour la classe `Child` car il est défini pour `Parent` , même s'il n'est pas défini explicitement dans la définition de classe de `Child` .

Dans cet exemple, la substitution se produit lorsque `Child` définit sa propre méthode `print_name` . Si cette méthode n'a pas été déclarée, alors `c.print_name()` aurait imprimé `"Parent"` . Toutefois, l' `Child` a la overriden `Parent` définition de l » `print_name` , et maintenant à appeler `c.print_name()` , le mot `"Child"` est imprimé.

Lire Dérogation de méthode en ligne: https://riptutorial.com/fr/python/topic/3131/derogation-de-methode

Chapitre 45: Des classes

Introduction

Python s'offre non seulement comme un langage de script populaire, mais prend également en charge le paradigme de programmation orientée objet. Les classes décrivent des données et fournissent des méthodes pour manipuler ces données, toutes regroupées sous un seul objet. De plus, les classes permettent l'abstraction en séparant les détails concrets de la mise en œuvre des représentations abstraites des données.

Les classes utilisant le code sont généralement plus faciles à lire, à comprendre et à gérer.

Examples

Héritage de base

L'héritage dans Python est basé sur des idées similaires utilisées dans d'autres langages orientés objet tels que Java, C ++, etc. Une nouvelle classe peut être dérivée d'une classe existante comme suit.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

La `BaseClass` est la classe (*parent*) existante et la classe `DerivedClass` est la nouvelle classe (*enfant*) qui hérite (ou *sous - classe*) les attributs de `BaseClass` . **Remarque** : à partir de Python 2.2, toutes les classes héritent implicitement de la classe d' `object` , qui est la classe de base de tous les types intégrés.

Nous définissons une classe `Rectangle` parent dans l'exemple ci-dessous, qui hérite implicitement de l' `object` :

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

La classe `Rectangle` peut être utilisée comme classe de base pour définir une classe `Square` , car un carré est un cas particulier de rectangle.

```
class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
        super(Square, self).__init__(s, s)
        self.s = s
```

La classe `Square` héritera automatiquement de tous les attributs de la classe `Rectangle` ainsi que de la classe d'objet. `super()` est utilisé pour appeler la `__init__()` de la classe `Rectangle` , appelant essentiellement toute méthode surchargée de la classe de base. **Note** : dans Python 3, `super()` ne nécessite aucun argument.

Les objets de classe dérivés peuvent accéder et modifier les attributs de ses classes de base:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

Fonctions intégrées qui fonctionnent avec l'héritage

`issubclass(DerivedClass, BaseClass)` : renvoie `True` si `DerivedClass` est une sous-classe de `BaseClass`

`isinstance(s, Class)` : renvoie `True` si s est une instance de `Class` ou de l'une des classes dérivées de `Class`

```
# subclass check
issubclass(Square, Rectangle)
# Output: True

# instantiate
r = Rectangle(3, 4)
s = Square(2)

isinstance(r, Rectangle)
# Output: True
isinstance(r, Square)
# Output: False
# A rectangle is not a square

isinstance(s, Rectangle)
# Output: True
# A square is a rectangle
isinstance(s, Square)
# Output: True
```

Variables de classe et d'instance

Les variables d'instance sont uniques pour chaque instance, tandis que les variables de classe sont partagées par toutes les instances.

```
class C:
    x = 2  # class variable

    def __init__(self, y):
        self.y = y  # instance variable

C.x
# 2
C.y
# AttributeError: type object 'C' has no attribute 'y'

c1 = C(3)
c1.x
# 2
c1.y
# 3

c2 = C(4)
c2.x
# 2
c2.y
# 4
```

Les variables de classe sont accessibles sur les instances de cette classe, mais l'affectation à l'attribut class créera une variable d'instance qui masque la variable de classe

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Notez que la *mutation* de variables de classe à partir d'instances peut entraîner des conséquences inattendues.

```
class D:
    x = []
    def __init__(self, item):
        self.x.append(item)  # note that this is not an assigment!

d1 = D(1)
d2 = D(2)

d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

Méthodes liées, non liées et statiques

L'idée des méthodes liées et non liées a été supprimée dans Python 3 . Dans Python 3, lorsque vous déclarez une méthode dans une classe, vous utilisez un mot-clé `def` , créant ainsi un objet fonction. Ceci est une fonction régulière et la classe environnante fonctionne comme son espace de noms. Dans l'exemple suivant, nous déclarons la méthode `f` dans la classe `A` , qui devient une fonction `Af` :

Python 3.x 3.0

```
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...>  (in Python 3.x)
```

Dans Python 2, le comportement était différent: les objets fonction de la classe étaient implicitement remplacés par des objets de type `instancemethod` , appelés *méthodes non liées,* car ils n'étaient liés à aucune instance de classe particulière. Il était possible d'accéder à la fonction sous-jacente en utilisant la propriété `.__func__` .

Python 2.x 2.3

```
A.f
# <unbound method A.f>   (in Python 2.x)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

Ces derniers comportements sont confirmés par inspection - les méthodes sont reconnues comme des fonctions dans Python 3, tandis que la distinction est confirmée dans Python 2.

Python 3.x 3.0

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Python 2.x 2.3

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

Dans les deux versions de Python, la fonction / méthode `Af` peut être appelée directement, à

condition que vous passiez une instance de classe `A` en premier argument.

```
A.f(1, 7)
# Python 2: TypeError: unbound method f() must be called with
#                      A instance as first argument (got int instance instead)
# Python 3: 14
a = A()
A.f(a, 20)
# Python 2 & 3: 40
```

Maintenant, supposons `a` est une instance de la classe `A` , qu'est-ce que `af` alors? Eh bien, intuitivement, cela devrait être la même méthode `f` de la classe `A` , mais elle devrait en quelque sorte "savoir" qu'elle a été appliquée à l'objet `a` - en Python, cela s'appelle la méthode *liée* à `a` .

Les détails de la débrouille sont les suivantes: l' écriture `af` invoque la magie `__getattribute__` méthode d' `a` , qui vérifie d' abord si `a` a un attribut nommé `f` (il n'a pas), puis vérifie la classe `A` si elle contient une méthode avec un tel nom (il le fait), et crée un nouvel objet `m` de type `method` qui a la référence à l' `Af` origine dans `m.__func__` , et une référence à l'objet `a` in `m.__self__` . Lorsque cet objet est appelé en tant que fonction, il fait simplement ce qui suit: `m(...) => m.__func__(m.__self__, ...)` . Ainsi, cet objet est appelé **méthode liée** car, lorsqu'il est appelé, il sait fournir l'objet auquel il était lié en tant que premier argument. (Ces choses fonctionnent de la même manière dans Python 2 et 3).

```
a = A()
a.f
# <bound method A.f of <__main__.A object at ...>>
a.f(2)
# 4

# Note: the bound method object a.f is recreated *every time* you call it:
a.f is a.f  # False
# As a performance optimization you can store the bound method in the object's
# __dict__, in which case the method object will remain fixed:
a.f = a.f
a.f is a.f  # True
```

Enfin, Python a **des méthodes de classe** et **des méthodes statiques** - des types spéciaux de méthodes. Les méthodes de classe fonctionnent de la même manière que les méthodes classiques, sauf que lorsqu'elles sont appelées sur un objet, elles sont liées à la *classe* de l'objet plutôt qu'à l'objet. Donc `m.__self__ = type(a)` . Lorsque vous appelez une telle méthode liée, elle passe la classe d' `a` comme premier argument. Les méthodes statiques sont encore plus simples: elles ne lient rien du tout et renvoient simplement la fonction sous-jacente sans aucune transformation.

```
class D(object):
    multiplier = 2

    @classmethod
    def f(cls, x):
        return cls.multiplier * x

    @staticmethod
    def g(name):
```

```
        print("Hello, %s" % name)

D.f
# <bound method type.f of <class '__main__.D'>>
D.f(12)
# 24
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

Notez que les méthodes de classe sont liées à la classe même si elles sont accessibles sur l'instance:

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
# (2, 1337)
d.f
# <bound method D.f of <class '__main__.D'>>
d.f(10)
# 20
```

Il est à noter qu'au niveau le plus bas, les fonctions, méthodes, méthodes statiques, etc. sont en réalité des descripteurs qui invoquent les méthodes spéciales `__get__` , `__set__` et éventuellement `__del__` . Pour plus de détails sur les méthodes de classe et les méthodes statiques:

- Quelle est la différence entre @staticmethod et @classmethod en Python?
- Signification de @classmethod et @staticmethod pour les débutants?

Nouveau style vs classes anciennes

Python 2.x 2.2.0

Les classes de *nouveau style* ont été introduites dans Python 2.2 pour unifier les *classes* et les *types* . Ils héritent du type d' `object` niveau supérieur. *Une classe de nouveau style est un type défini par l'utilisateur* et très similaire aux types intégrés.

```
# new-style class
class New(object):
    pass

# new-style instance
new = New()

new.__class__
# <class '__main__.New'>
type(new)
# <class '__main__.New'>
issubclass(New, object)
# True
```

Les anciennes classes n'héritent **pas** de l' `object` . Les anciennes instances sont toujours implémentées avec un type d' `instance` intégré.

```
# old-style class
class Old:
    pass

# old-style instance
old = Old()

old.__class__
# <class __main__.Old at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Python 3.x 3.0.0

En Python 3, les anciennes classes ont été supprimées.

Les classes de nouveau style dans Python 3 héritent implicitement de l' `object` , il n'est donc plus nécessaire de spécifier `MyClass(object)` .

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

Valeurs par défaut pour les variables d'instance

Si la variable contient une valeur d'un type immuable (par exemple une chaîne), il est possible d'attribuer une valeur par défaut comme celle-ci

```
class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width  * self.height

# Create some instances of the class
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # blue

red_rectangle = Rectangle(2, 3, 'red')
print(red_rectangle.color) # red
```

Il faut faire attention lors de l'initialisation d'objets mutables tels que les listes dans le constructeur. Prenons l'exemple suivant:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=[0,0], color='blue'):
        self.width = width
        self.height = height
        self.pos = pos
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [4, 0] r2's pos has changed as well
```

Ce comportement est dû au fait que dans Python, les paramètres par défaut sont liés à l'exécution de la fonction et non à la déclaration de la fonction. Pour obtenir une variable d'instance par défaut qui n'est pas partagée entre les instances, il faut utiliser une construction comme celle-ci:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # default value is [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [0, 0] r2's pos hasn't changed
```

Voir également les arguments de Mutable par défaut et la «moindre surprise» et l'argument par défaut Mutable .

Héritage multiple

Python utilise l'algorithme de linéarisation C3 pour déterminer l'ordre de résolution des attributs de classe, y compris les méthodes. Ceci est connu comme l'Ordre de Résolution de Méthode (MRO).

Voici un exemple simple:

```
class Foo(object):
    foo = 'attr foo of Foo'


class Bar(object):
    foo = 'attr foo of Bar' # we won't see this.
    bar = 'attr bar of Bar'

class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'
```

Maintenant, si nous instancions FooBar, si nous recherchons l'attribut foo, nous voyons que l'attribut de Foo est trouvé en premier

```
fb = FooBar()
```

et

```
>>> fb.foo
'attr foo of Foo'
```

Voici le MRO de FooBar:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

On peut simplement dire que l'algorithme MRO de Python est

1. Profondeur d'abord (par exemple `FooBar` puis `Foo`) à moins que
2. un parent partagé (`object`) est bloqué par un enfant (`Bar`) et
3. aucune relation circulaire autorisée.

C'est-à-dire que, par exemple, Bar ne peut pas hériter de FooBar alors que FooBar hérite de Bar.

Pour un exemple complet en Python, voir l' entrée Wikipedia .

Une autre fonctionnalité puissante dans l'héritage est `super` . super peut récupérer les fonctionnalités des classes parentes.

```
class Foo(object):
    def foo_method(self):
        print "foo Method"

class Bar(object):
    def bar_method(self):
        print "bar Method"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

Héritage multiple avec la méthode init de la classe, lorsque chaque classe a sa propre méthode d'initialisation, alors nous essayons d'inférencer plusieurs fois, alors seule la méthode init est appelée de classe qui est héritée en premier.

pour exemple ci - dessous seule méthode **init** Foo classe se classe appelé **Bar** INIT pas appelé

```
    class Foo(object):
        def __init__(self):
            print "foo init"

    class Bar(object):
        def __init__(self):
            print "bar init"

    class FooBar(Foo, Bar):
        def __init__(self):
```

```
            print "foobar init"
            super(FooBar, self).__init__()

    a = FooBar()
```

Sortie:

```
    foobar init
    foo init
```

Mais cela ne signifie pas que la classe **Bar** n'est pas héritée. L'instance de la classe **FooBar** finale est également une instance de la classe **Bar** et de la classe **Foo** .

```
print isinstance(a,FooBar)
print isinstance(a,Foo)
print isinstance(a,Bar)
```

Sortie:

```
True
True
True
```

Descripteurs et recherches par points

Les descripteurs sont des objets qui sont (généralement) des attributs de classes et qui ont des méthodes spéciales `__get__` , `__set__` ou `__delete__` .

Les descripteurs de données ont l'un des `__set__` ou `__delete__`

Ceux-ci peuvent contrôler la recherche pointillée sur une instance et sont utilisés pour implémenter des fonctions, `staticmethod` , `classmethod` et `property` . Une recherche pointillée (par exemple, instance `foo` de la classe `Foo` recherchant la `bar` attributs - c.-à-d. `foo.bar`) utilise l'algorithme suivant:

1. `bar` est levé dans la classe, `Foo` . S'il existe et qu'il s'agit d'un **descripteur de données** , le descripteur de données est utilisé. C'est comme cela que la `property` peut contrôler l'accès aux données dans une instance et que les instances ne peuvent pas remplacer cela. Si un **descripteur de données** n'est pas là, alors

2. `bar` est recherché dans l'instance `__dict__` . C'est pourquoi nous pouvons remplacer ou bloquer les méthodes appelées à partir d'une instance avec une recherche pointillée. Si la `bar` existe dans l'instance, elle est utilisée. Sinon, nous avons alors

3. regardez dans la classe `Foo` for `bar` . S'il s'agit d'un **descripteur** , le protocole de descripteur est utilisé. Voici comment les fonctions (dans ce contexte, les méthodes non liées), `classmethod` et `staticmethod` sont implémentées. Sinon, il retourne simplement l'objet, ou il y a un `AttributeError`

Méthodes de classe: initialiseurs alternatifs

Les méthodes de classe présentent des méthodes alternatives pour générer des instances de classes. Pour illustrer, regardons un exemple.

Supposons que nous ayons une classe de `Person` relativement simple:

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Il pourrait être utile d'avoir un moyen de construire des instances de cette classe en spécifiant un nom complet au lieu du nom et du prénom séparément. Une façon de le faire serait de faire en sorte que `last_name` soit un paramètre facultatif, et en supposant que s'il n'est pas donné, nous avons passé le nom complet dans:

```
class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Cependant, il y a deux problèmes principaux avec ce bit de code:

1. Les paramètres `first_name` et `last_name` sont maintenant trompeuses, puisque vous pouvez entrer un nom complet pour `first_name` . En outre, s'il y a plus de cas et / ou de paramètres ayant ce type de flexibilité, le branchement if / elif / else peut devenir rapidement ennuyeux.
2. Pas tout à fait aussi important, mais ça vaut quand même la peine de le souligner: et si `last_name` est `None` , mais `first_name` ne se divise pas en deux ou plus via des espaces? Nous avons encore une autre couche de validation des entrées et / ou de gestion des exceptions ...

Entrez les méthodes de classe. Plutôt que d'avoir un seul initialiseur, nous allons créer un initialiseur distinct, appelé `from_full_name` , et le décorer avec le décorateur de `classmethod` (intégré).

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    @classmethod
    def from_full_name(cls, name, age):
        if " " not in name:
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        return cls(first_name, last_name, age)

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Remarquez `cls` au lieu de `self` comme premier argument de `from_full_name` . Les méthodes de classe sont appliquées à la classe globale, et *non à* une instance d'une classe donnée (ce que `self` désigne généralement). Donc, si `cls` est notre `Person` de classe, la valeur renvoyée par la `from_full_name` méthode de classe est `Person(first_name, last_name, age)` , qui utilise la `Person` de `__init__` pour créer une instance de la `Person` classe. En particulier, si nous devions créer une sous-classe `Employee` of `Person` , alors `from_full_name` fonctionnerait également dans la classe `Employee` .

Pour montrer que cela fonctionne comme prévu, créons des instances de `Person` de plusieurs manières sans les branchements dans `__init__` :

```
In [2]: bob = Person("Bob", "Bobberson", 42)

In [3]: alice = Person.from_full_name("Alice Henderson", 31)

In [4]: bob.greet()
Hello, my name is Bob Bobberson.

In [5]: alice.greet()
Hello, my name is Alice Henderson.
```

Autres références:

- Python @classmethod et @staticmethod pour les débutants?
- https://docs.python.org/2/library/functions.html#classmethod
- https://docs.python.org/3.5/library/functions.html#classmethod

Composition de classe

La composition de classe permet des relations explicites entre les objets. Dans cet exemple, les gens vivent dans des villes appartenant à des pays. La composition permet aux personnes d'accéder au nombre de personnes vivant dans leur pays:

```
class Country(object):
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)


class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople


    def addPerson(self, person):
        self.people.append(person)

    def join_country(self,country):
        self.country = country
        country.addCity(self)

        for i in range(self.numPeople):
                person(i).join_city(self)


class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x

US=Country()
NYC=City(10).join_country(US)
SF=City(5).join_country(US)

print(US.cities[0].people[0].people_in_my_country())

# 15
```

Singe Patching

Dans ce cas, "patching singe" signifie l'ajout d'une nouvelle variable ou méthode à une classe après sa définition. Par exemple, disons que nous avons défini la classe `A` comme

```
class A(object):
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

Mais maintenant, nous voulons ajouter une autre fonction plus tard dans le code. Supposons que

cette fonction est la suivante.

```
def get_num(self):
    return self.num
```

Mais comment ajouter cela comme méthode dans `A` ? C'est simple, nous plaçons essentiellement cette fonction dans `A` avec une déclaration d'affectation.

```
A.get_num = get_num
```

Pourquoi ça marche? Parce que les fonctions sont des objets comme n'importe quel autre objet, et que les méthodes sont des fonctions appartenant à la classe.

La fonction `get_num` doit être disponible pour tous les existants (déjà créés) ainsi que pour les nouvelles instances de `A`

Ces ajouts sont disponibles automatiquement sur toutes les instances de cette classe (ou de ses sous-classes). Par exemple:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42

bar.get_num() # 6
```

Notez que, contrairement à d'autres langages, cette technique ne fonctionne pas pour certains types intégrés et n'est pas considérée comme un bon style.

Liste de tous les membres de la classe

La fonction `dir()` peut être utilisée pour obtenir une liste des membres d'une classe:

```
dir(Class)
```

Par exemple:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Il est courant de ne chercher que des membres "non magiques". Cela peut être fait en utilisant une compréhension simple qui répertorie les membres dont le nom ne commence pas par __ :

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Mises en garde:

Les classes peuvent définir une `__dir__()` . Si cette méthode existe, appeler `dir()` appellera `__dir__()` , sinon Python essaiera de créer une liste des membres de la classe. Cela signifie que la fonction dir peut avoir des résultats inattendus. Deux citations importantes de la documentation officielle de python :

> Si l'objet ne fournit pas **dir** (), la fonction essaie de son mieux de collecter des informations à partir de l'attribut **dict** de l'objet, s'il est défini, et de son objet de type. La liste résultante n'est pas nécessairement complète et peut être inexacte lorsque l'objet a un **getattr** () personnalisé.
>
> **Remarque:** Étant donné que rép () est principalement utilisé pour une utilisation sur une invite interactive, il essaie de fournir un ensemble de noms intéressant plus qu'il ne tente de fournir un ensemble de noms rigoureusement ou systématiquement défini, et son comportement détaillé peut varier rejets. Par exemple, les attributs de métaclasse ne sont pas dans la liste de résultats lorsque l'argument est une classe.

Introduction aux cours

Une classe fonctionne comme un modèle définissant les caractéristiques de base d'un objet particulier. Voici un exemple:

```
class Person(object):
     """A simple class."""                              # docstring
     species = "Homo Sapiens"                           # class attribute

     def __init__(self, name):                          # special method
         """This is the initializer. It's a special
         method (see below)."""
         self.name = name                               # instance attribute

     def __str__(self):                                 # special method
         """This method is run when Python tries
         to cast the object to a string. Return
         this string when using print(), etc."""
         return self.name

     def rename(self, renamed):                         # regular method
         """Reassign and print the name attribute."""
         self.name = renamed
         print("Now my name is {}".format(self.name))
```

Il y a quelques points à noter lorsque vous regardez l'exemple ci-dessus.

1. La classe est composée d' *attributs* (données) et de *méthodes* (fonctions).
2. Les attributs et les méthodes sont simplement définis comme des variables et des fonctions

normales.

3. Comme indiqué dans le docstring correspondant, la `__init__()` s'appelle l' *initialiseur* . C'est l'équivalent du constructeur dans d'autres langages orientés objet, et c'est la méthode qui est exécutée en premier lorsque vous créez un nouvel objet ou une nouvelle instance de la classe.
4. Les attributs qui s'appliquent à la classe entière sont définis en premier et sont appelés *attributs de classe* .
5. Les attributs qui s'appliquent à une instance spécifique d'une classe (un objet) sont appelés *attributs d'instance* . Ils sont généralement définis dans `__init__()` ; ce n'est pas nécessaire, mais cela est recommandé (puisque les attributs définis en dehors de `__init__()` risquent d'être consultés avant d'être définis).
6. Chaque méthode, incluse dans la définition de classe, passe l'objet en question comme premier paramètre. Le mot `self` est utilisé pour ce paramètre (l'utilisation de `self` est en réalité par convention, car le mot `self` n'a aucune signification inhérente en Python, mais c'est l'une des conventions les plus respectées de Python, et vous devriez toujours la suivre).
7. Ceux qui sont habitués à la programmation orientée objet dans d'autres langues peuvent être surpris par certaines choses. L'une d'elles est que Python n'a pas de véritable concept d'éléments `private` , donc tout, par défaut, imite le comportement du mot clé `public` C ++ / Java. Pour plus d'informations, reportez-vous à l'exemple "Private Class Members" sur cette page.
8. Certaines des méthodes de la classe ont la forme suivante: `__functionname__(self, other_stuff)` . Toutes ces méthodes sont appelées "méthodes magiques" et constituent une partie importante des classes en Python. Par exemple, la surcharge de l'opérateur en Python est implémentée avec des méthodes magiques. Pour plus d'informations, consultez la documentation appropriée .

Faisons maintenant quelques exemples de notre classe `Person` !

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

Nous avons actuellement trois objets `Person` , `kelly` , `joseph` et `john_doe` .

Nous pouvons accéder aux attributs de la classe à partir de chaque instance à l'aide de l'opérateur point . Notez à nouveau la différence entre les attributs de classe et d'instance:

```
>>> # Attributes
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

Nous pouvons exécuter les méthodes de la classe en utilisant le même opérateur de points . :

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>>  print(john_doe)
'John Doe'
>>>  john_doe.rename("John")
'Now my name is John'
```

Propriétés

Les classes Python prennent en charge les **propriétés** , qui ressemblent à des variables d'objet classiques, mais avec la possibilité d'attacher un comportement et une documentation personnalisés.

```
class MyClass(object):

    def __init__(self):
       self._my_string = ""

    @property
    def string(self):
        """A profoundly important string."""
        return self._my_string

    @string.setter
    def string(self, new_value):
        assert isinstance(new_value, str), \
               "Give me a string, not a %r!" % type(new_value)
        self._my_string = new_value

    @string.deleter
    def x(self):
        self._my_string = None
```

L'objet de la classe `MyClass` *semblera* avoir une propriété `.string` , mais son comportement est maintenant étroitement contrôlé:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

Outre la syntaxe utile ci-dessus, la syntaxe de la propriété permet la validation ou d'autres augmentations à ajouter à ces attributs. Cela pourrait être particulièrement utile avec les API publiques - où un niveau d'aide devrait être donné à l'utilisateur.

Une autre utilisation courante des propriétés est de permettre à la classe de présenter des «attributs virtuels» - des attributs qui ne sont pas réellement stockés mais qui ne sont calculés que sur demande.

```
class Character(object):
```

```
    def __init__(name, max_hp):
        self._name = name
        self._hp = max_hp
        self._max_hp = max_hp

    # Make hp read only by not providing a set method
    @property
    def hp(self):
        return self._hp

    # Make name read only by not providing a set method
    @property
    def name(self):
        return self.name

    def take_damage(self, damage):
        self.hp -= damage
        self.hp = 0 if self.hp <0 else self.hp

    @property
    def is_alive(self):
        return self.hp != 0

    @property
    def is_wounded(self):
        return self.hp < self.max_hp if self.hp > 0 else False

    @property
    def is_dead(self):
        return not self.is_alive

bilbo = Character('Bilbo Baggins', 100)
bilbo.hp
# out : 100
bilbo.hp = 200
# out : AttributeError: can't set attribute
# hp attribute is read only.

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )

bilbo.hp
# out : 50

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : True
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )
bilbo.hp
# out : 0
```

```
bilbo.is_alive
# out : False
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : True
```

Classe Singleton

Un singleton est un modèle qui limite l'instanciation d'une classe à une instance / objet. Pour plus d'informations sur les modèles de conception de singleton python, voir ici .

```
class Singleton:
    def __new__(cls):
        try:
            it = cls.__it__
        except AttributeError:
            it = cls.__it__ = object.__new__(cls)
        return it

    def __repr__(self):
        return '<{}>'.format(self.__class__.__name__.upper())

    def __eq__(self, other):
         return other is self
```

Une autre méthode consiste à décorer votre classe. En suivant l'exemple de cette réponse, créez une classe Singleton:

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.

    The decorated class can define one `__init__` function that
    takes only the `self` argument. Other than that, there are
    no restrictions that apply to the decorated class.

    To get the singleton instance, use the `Instance` method. Trying
    to use `__call__` will result in a `TypeError` being raised.

    Limitations: The decorated class cannot be inherited from.

    """

    def __init__(self, decorated):
        self._decorated = decorated

    def Instance(self):
        """
        Returns the singleton instance. Upon its first call, it creates a
        new instance of the decorated class and calls its `__init__` method.
        On all subsequent calls, the already created instance is returned.

        """
        try:
```

```
            return self._instance
        except AttributeError:
            self._instance = self._decorated()
            return self._instance

    def __call__(self):
        raise TypeError('Singletons must be accessed through `Instance()`.')

    def __instancecheck__(self, inst):
        return isinstance(inst, self._decorated)
```

Pour utiliser, vous pouvez utiliser la méthode `Instance`

```
@Singleton
class Single:
    def __init__(self):
        self.name=None
        self.val=0
    def getName(self):
        print(self.name)

x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # outputs I'm single
y.getName() # outputs I'm single
```

Lire Des classes en ligne: https://riptutorial.com/fr/python/topic/419/des-classes

Chapitre 46: Des exceptions

Introduction

Les erreurs détectées lors de l'exécution s'appellent des exceptions et ne sont pas inconditionnellement fatales. La plupart des exceptions ne sont pas gérées par les programmes. il est possible d'écrire des programmes qui traitent les exceptions sélectionnées. Il existe des fonctionnalités spécifiques à Python pour gérer les exceptions et la logique des exceptions. De plus, les exceptions ont une hiérarchie de types enrichis, toutes héritant du type `BaseException` .

Syntaxe

- lever l' *exception*
- raise # re-relance une exception qui a déjà été soulevée
- *déclenche une exception* de *cause* # Python 3 - définit la cause des exceptions
- déclenche une *exception à* partir de None # Python 3 - supprime tous les contextes d'exception
- essayer:
- sauf *[types d'exception] [* comme *identifiant]* :
- autre:
- enfin:

Examples

Augmenter les exceptions

Si votre code rencontre une condition qu'il ne sait pas gérer, telle qu'un paramètre incorrect, il doit déclencher l'exception appropriée.

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an odd number")

    return odds + 1
```

Prendre des exceptions

Utilisez `try...except:` pour attraper des exceptions. Vous devez spécifier une exception aussi précise que possible:

```
try:
    x = 5 / 0
except ZeroDivisionError as e:
    # `e` is the exception object
    print("Got a divide by zero! The exception was:", e)
    # handle exceptional case
```

```
    x = 0
finally:
    print "The END"
    # it runs no matter what execute.
```

La classe d'exception spécifiée - dans ce cas, `ZeroDivisionError` - `ZeroDivisionError` toute exception `ZeroDivisionError` cette classe ou à une sous-classe de cette exception.

Par exemple, `ZeroDivisionError` est une sous-classe de `ArithmeticError` :

```
>>> ZeroDivisionError.__bases__
(<class 'ArithmeticError'>,)
```

Et ainsi, les éléments suivants attraperont toujours le `ZeroDivisionError` :

```
try:
    5 / 0
except ArithmeticError:
    print("Got arithmetic error")
```

Lancer le code de nettoyage avec finalement

Parfois, vous voudrez peut-être que quelque chose se produise, quelle que soit l'exception, par exemple, si vous devez nettoyer certaines ressources.

Le bloc `finally` d'une clause `try` se produira, que des exceptions aient été soulevées ou non.

```
resource = allocate_some_expensive_resource()
try:
    do_stuff(resource)
except SomeException as e:
    log_error(e)
    raise  # re-raise the error
finally:
    free_expensive_resource(resource)
```

Ce modèle est souvent mieux géré avec les gestionnaires de contexte (en utilisant l'instruction `with`).

Relancer les exceptions

Parfois, vous voulez attraper une exception simplement pour l'inspecter, par exemple à des fins de journalisation. Après l'inspection, vous souhaitez que l'exception continue à se propager comme auparavant.

Dans ce cas, utilisez simplement l'instruction `raise` sans paramètres.

```
try:
    5 / 0
except ZeroDivisionError:
    print("Got an error")
```

```
    raise
```

Gardez à l'esprit que quelqu'un plus haut dans la pile de l'appelant peut toujours attraper l'exception et la gérer d'une manière ou d'une autre. Le résultat final pourrait être une nuisance dans ce cas car cela se produira dans tous les cas (attrapé ou non attrapé). Donc, il serait peut-être préférable de soulever une exception différente, contenant votre commentaire sur la situation ainsi que l'exception initiale:

```
try:
    5 / 0
except ZeroDivisionError as e:
    raise ZeroDivisionError("Got an error", e)
```

Mais cela a l'inconvénient de réduire la trace d'exception à exactement cette `raise` tandis que `raise` sans argument conserve la trace d'exception d'origine.

En Python 3, vous pouvez conserver la pile d'origine en utilisant la syntaxe `raise` - `from` :

```
    raise ZeroDivisionError("Got an error") from e
```

Chaîne d'exceptions avec augmentation de

Lors du traitement d'une exception, vous souhaiterez peut-être générer une autre exception. Par exemple, si vous obtenez une `IOError` lors de la lecture d'un fichier, vous souhaiterez peut-être générer une erreur spécifique à l'application pour la présenter aux utilisateurs de votre bibliothèque.

Python 3.x 3.0

Vous pouvez enchaîner les exceptions pour montrer comment le traitement des exceptions s'est déroulé:

```
>>> try:
    5 / 0
except ZeroDivisionError as e:
    raise ValueError("Division failed") from e

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: Division failed
```

Hiérarchie des exceptions

La gestion des exceptions se produit en fonction d'une hiérarchie d'exceptions, déterminée par la structure d'héritage des classes d'exception.

Par exemple, `IOError` et `OSError` sont deux sous-classes de `EnvironmentError` . Le code qui `IOError` une `IOError` pas une `OSError` . Toutefois, le code qui intercepte une `EnvironmentError` intercepte à la fois `IOError` s et `OSError` s.

La hiérarchie des exceptions intégrées:

Python 2.x 2.3

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
       +-- ImportWarning
       +-- UnicodeWarning
       +-- BytesWarning
```

Python 3.x 3.0

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
```

```
          +-- BytesWarning
          +-- ResourceWarning
```

Les exceptions sont des objets aussi

Les exceptions ne sont que des objets Python classiques qui héritent de `BaseException` . Un script Python peut utiliser l'instruction `raise` pour interrompre l'exécution, entraînant Python à imprimer une trace de pile de la pile d'appels à ce stade et une représentation de l'instance d'exception. Par exemple:

```
>>> def failing_function():
...     raise ValueError('Example error!')
>>> failing_function()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in failing_function
ValueError: Example error!
```

qui dit qu'un `ValueError` avec le message `'Example error!'` a été `failing_function()` par notre `failing_function()` , qui a été exécuté dans l'interpréteur.

Le code d'appel peut choisir de gérer tous les types d'exception qu'un appel peut générer:

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Handled the error')
Handled the error
```

Vous pouvez mettre la main sur les objets d'exception en les affectant dans la partie `except...` du code de gestion des exceptions:

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Caught exception', repr(e))
Caught exception ValueError('Example error!',)
```

Une liste complète des exceptions Python intégrées avec leurs descriptions se trouve dans la documentation Python: https://docs.python.org/3.5/library/exceptions.html . Et voici la liste complète hiérarchisée: Hiérarchie des exceptions .

Création de types d'exception personnalisés

Créez une classe héritant d' `Exception` :

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
except FooException:
```

```
    print("A FooException was raised.")
```

ou un autre type d'exception:

```
class NegativeError(ValueError):
      pass

def foo(x):
    # function that only accepts positive values of x
    if x < 0:
        raise NegativeError("Cannot process negative numbers")
    ...  # rest of function body
try:
    result = foo(int(input("Enter a positive integer: ")))  # raw_input in Python 2.x
except NegativeError:
    print("You entered a negative number!")
else:
    print("The result was " + str(result))
```

Ne pas attraper tout!

Bien qu'il soit souvent tentant d'attraper chaque `Exception` :

```
try:
    very_difficult_function()
except Exception:
    # log / try to reconnect / exit gratiously
finally:
    print "The END"
    # it runs no matter what execute.
```

Ou même tout ce qui inclut `BaseException` et tous ses enfants, y compris `Exception`):

```
try:
    even_more_difficult_function()
except:
    pass  # do whatever needed
```

Dans la plupart des cas, c'est une mauvaise pratique. Cela peut prendre plus de temps que prévu, comme `SystemExit` , `KeyboardInterrupt` et `MemoryError` , chacun d'entre eux devant généralement être traité différemment des erreurs système ou logiques habituelles. Cela signifie également qu'il n'y a pas de compréhension claire de ce que le code interne peut faire de mal et comment récupérer correctement de cette condition. Si vous attrapez toutes les erreurs, vous ne saurez pas quelle erreur s'est produite ou comment y remédier.

Ceci est plus communément appelé «masquage de bogue» et devrait être évité. Laissez votre programme tomber en panne au lieu d'échouer silencieusement ou pire encore, en échouant à un niveau d'exécution plus profond. (Imaginez que c'est un système transactionnel)

Habituellement, ces constructions sont utilisées au niveau le plus externe du programme et enregistrent les détails de l'erreur afin que le bogue puisse être corrigé ou que l'erreur soit traitée de manière plus spécifique.

Prendre plusieurs exceptions

Il existe plusieurs moyens de détecter les exceptions multiples .

La première consiste à créer un tuple des types d'exception que vous souhaitez capturer et gérer de la même manière. Cet exemple entraînera le code à ignorer les exceptions `KeyError` et `AttributeError` .

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except (KeyError, AttributeError) as e:
    print("A KeyError or an AttributeError exception has been caught.")
```

Si vous souhaitez gérer différentes exceptions de différentes manières, vous pouvez fournir un bloc d'exception distinct pour chaque type. Dans cet exemple, nous interceptons toujours `KeyError` et `AttributeError` , mais gérons les exceptions de différentes manières.

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except KeyError as e:
    print("A KeyError has occurred. Exception message:", e)
except AttributeError as e:
    print("An AttributeError has occurred. Exception message:", e)
```

Exemples pratiques de gestion des exceptions

Entrée utilisateur

Imaginez que vous souhaitiez qu'un utilisateur entre un numéro via une `input` . Vous voulez vous assurer que l'entrée est un nombre. Vous pouvez utiliser `try` / `except` ceci:

Python 3.x 3.0

```
while True:
    try:
        nb = int(input('Enter a number: '))
        break
    except ValueError:
        print('This is not a number, try again.')
```

Remarque: Python 2.x utiliserait plutôt `raw_input` ; la fonction `input` existe dans Python 2.x mais a une sémantique différente. Dans l'exemple ci-dessus, `input` accepterait également des expressions telles que `2 + 2` qui évaluent un nombre.

Si l'entrée n'a pas pu être convertie en entier, une valeur `ValueError` est `ValueError` . Vous pouvez

l'attraper avec, `except` . Si aucune exception est levée, la `break` saute hors de la boucle. Après la boucle, `nb` contient un entier.

Dictionnaires

Imaginez que vous itérez une liste d'entiers consécutifs, comme `range(n)` , et vous avez une liste de dictionnaires `d` qui contient des informations sur les choses à faire lorsque vous rencontrez des entiers particuliers, dites *sauter le `d[i]` ones suivantes.*

```
d = [{7: 3}, {25: 9}, {38: 5}]

for i in range(len(d)):
    do_stuff(i)
    try:
       dic = d[i]
       i += dic[i]
    except KeyError:
       i += 1
```

Un `KeyError` sera `KeyError` lorsque vous essayez d'obtenir une valeur d'un dictionnaire pour une clé qui n'existe pas.

Autre

Le code dans un bloc else ne sera exécuté que si aucune exception n'a été déclenchée par le code dans le bloc `try` . Ceci est utile si vous ne voulez pas exécuter du code si une exception est levée, mais vous ne voulez pas que les exceptions lancées par ce code soient interceptées.

Par exemple:

```
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# Output: one
# Output: ValueError
```

Notez que ce genre d' `else:` ne peut pas être combiné avec un `if` la clause `elif` dans un `elif` . Si vous avez un public `if` elle doit rester en retrait en dessous d' `else:` :

```
try:
    ...
except ...:
    ...
else:
    if ...:
        ...
    elif ...:
```

```
        ...
    else:
        ...
```

Lire Des exceptions en ligne: https://riptutorial.com/fr/python/topic/1788/des-exceptions

Chapitre 47: Descripteur

Examples

Descripteur simple

Il existe deux types de descripteurs différents. Les descripteurs de données sont définis comme des objets définissant à la fois une `__get__()` et une `__set__()` , tandis que les descripteurs autres que des données définissent uniquement une `__get__()` . Cette distinction est importante lorsque l'on considère les substitutions et l'espace de noms du dictionnaire d'une instance. Si un descripteur de données et une entrée du dictionnaire d'une instance partagent le même nom, le descripteur de données aura la priorité. Cependant, si à la place un descripteur de non-données et une entrée du dictionnaire d'une instance partagent le même nom, l'entrée du dictionnaire d'instance est prioritaire.

Pour créer un descripteur de données en lecture seule, définissez à la fois **get** () et **set** () avec **set** () en levant un AttributeError lorsqu'il est appelé. Définir la méthode **set** () avec un espace réservé de levée d'exception suffit pour en faire un descripteur de données.

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Un exemple implémenté:

```
class DescPrinter(object):
    """A data descriptor that logs activity."""
    _val = 7

    def __get__(self, obj, objtype=None):
        print('Getting ...')
        return self._val

    def __set__(self, obj, val):
        print('Setting', val)
        self._val = val

    def __delete__(self, obj):
        print('Deleting ...')
        del self._val


class Foo():
    x = DescPrinter()

i = Foo()
i.x
# Getting ...
# 7

i.x = 100
# Setting 100
```

```
i.x
# Getting ...
# 100

del i.x
# Deleting ...
i.x
# Getting ...
# 7
```

Conversions bidirectionnelles

Les objets descripteurs peuvent permettre aux attributs d'objet associés de réagir automatiquement aux modifications.

Supposons que nous voulions modéliser un oscillateur avec une fréquence donnée (en Hertz) et une période (en secondes). Lorsque nous mettons à jour la fréquence à laquelle nous souhaitons mettre à jour la période et que nous mettons à jour la période, nous voulons que la fréquence soit mise à jour:

```
 >>> oscillator = Oscillator(freq=100.0)  # Set frequency to 100.0 Hz
>>> oscillator.period  # Period is 1 / frequency, i.e. 0.01 seconds
0.01
>>> oscillator.period = 0.02  # Set period to 0.02 seconds
>>> oscillator.freq # The frequency is automatically adjusted
50.0
>>> oscillator.freq = 200.0  # Set the frequency to 200.0 Hz
>>> oscillator.period  # The period is automatically adjusted
0.005
```

Nous sélectionnons l'une des valeurs (fréquence, en Hertz) comme "ancre", c'est-à-dire celle qui peut être définie sans conversion, et écrivons une classe de descripteur pour cela:

```
class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)
```

La valeur "autre" (point, en secondes) est définie en fonction de l'ancre. Nous écrivons une classe de descripteur qui effectue nos conversions:

```
class Second(object):
    def __get__(self, instance, owner):
        # When reading period, convert from frequency
        return 1 / instance.freq

    def __set__(self, instance, value):
        # When setting period, update the frequency
        instance.freq = 1 / float(value)
```

Nous pouvons maintenant écrire la classe Oscillator:

```
class Oscillator(object):
    period = Second()  # Set the other value as a class attribute

    def __init__(self, freq):
        self.freq = Hertz()  # Set the anchor value as an instance attribute
        self.freq = freq  # Assign the passed value - self.period will be adjusted
```

Lire Descripteur en ligne: https://riptutorial.com/fr/python/topic/3405/descripteur

Chapitre 48: Déstructurer la liste (aka emballage et déballage)

Examples

Affectation de destruction

Dans les affectations, vous pouvez diviser une Iterable en valeurs en utilisant la syntaxe "unpacking":

Destructuration en tant que valeurs

```
a, b = (1, 2)
print(a)
# Prints: 1
print(b)
# Prints: 2
```

Si vous essayez de décompresser plus que la longueur de l'itérable, vous obtenez une erreur:

```
a, b, c = [1]
# Raises: ValueError: not enough values to unpack (expected 3, got 1)
```

Python 3.x 3.0

Destructuration en liste

Vous pouvez décompresser une liste de longueur inconnue en utilisant la syntaxe suivante:

```
head, *tail = [1, 2, 3, 4, 5]
```

Ici, nous extrayons la première valeur en tant que scalaire et les autres valeurs en tant que liste:

```
print(head)
# Prints: 1
print(tail)
# Prints: [2, 3, 4, 5]
```

Ce qui équivaut à:

```
l = [1, 2, 3, 4, 5]
head = l[0]
tail = l[1:]
```

Il fonctionne également avec plusieurs éléments ou éléments à la fin de la liste:

```
a, b, *other, z = [1, 2, 3, 4, 5]
print(a, b, z, other)
# Prints: 1 2 5 [3, 4]
```

Ignorer les valeurs dans les affectations de déstructuration

Si vous êtes seulement intéressé par une valeur donnée, vous pouvez utiliser _ pour indiquer que vous n'êtes pas intéressé. Remarque: ceci sera toujours défini _ , la plupart des gens ne l'utilisent pas comme variable.

```
a, _ = [1, 2]
print(a)
# Prints: 1
a, _, c = (1, 2, 3)
print(a)
# Prints: 1
print(c)
# Prints: 3
```

Python 3.x 3.0

Ignorer les listes dans les affectations de déstructuration

Enfin, vous pouvez ignorer plusieurs valeurs en utilisant la syntaxe `*_` dans l’affectation:

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Prints: 1
```

ce qui n'est pas vraiment intéressant, car vous pouvez utiliser l'indexation sur la liste à la place. Il est bon de garder les premières et les dernières valeurs en une seule tâche:

```
a, *_, b = [1, 2, 3, 4, 5]
print(a, b)
# Prints: 1 5
```

ou extrayez plusieurs valeurs à la fois:

```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
print(a, b, c)
# Prints: 1 3 5
```

Arguments de la fonction d'emballage

Dans les fonctions, vous pouvez définir un certain nombre d'arguments obligatoires:

```
def fun1(arg1, arg2, arg3):
    return (arg1,arg2,arg3)
```

qui rendra la fonction appelable uniquement lorsque les trois arguments sont donnés:

```
fun1(1, 2, 3)
```

et vous pouvez définir les arguments comme facultatifs, en utilisant les valeurs par défaut:

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1,arg2,arg3)
```

vous pouvez donc appeler la fonction de différentes manières, par exemple:

```
fun2(1)              → (1,b,c)
fun2(1, 2)           → (1,2,c)
fun2(arg2=2, arg3=3) → (a,2,3)
...
```

Mais vous pouvez aussi utiliser la syntaxe déstructurant pour *emballer* des arguments, donc vous pouvez assigner des variables à l' aide d' une `list` ou d' un `dict` .

Emballage d'une liste d'arguments

Considérez que vous avez une liste de valeurs

```
l = [1,2,3]
```

Vous pouvez appeler la fonction avec la liste de valeurs en tant qu'argument en utilisant la syntaxe `*` :

```
fun1(*l)
# Returns: (1,2,3)
fun1(*['w', 't', 'f'])
# Returns: ('w','t','f')
```

Mais si vous ne fournissez pas une liste dont la longueur correspond au nombre d'arguments:

```
fun1(*['oops'])
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3'
```

Arguments sur les mots-clés d'emballage

Maintenant, vous pouvez également empaqueter des arguments en utilisant un dictionnaire. Vous pouvez utiliser l'opérateur `**` pour demander à Python de décompresser le `dict` tant que valeurs de paramètre:

```
d = {
  'arg1': 1,
  'arg2': 2,
  'arg3': 3
```

```
}
fun1(**d)
# Returns: (1, 2, 3)
```

Lorsque la fonction ne contient que des arguments de position (ceux sans valeur par défaut), vous avez besoin que le dictionnaire contienne tous les paramètres attendus et qu'il n'y ait pas de paramètre supplémentaire, sinon vous obtiendrez une erreur:

```
fun1(**{'arg1':1, 'arg2':2})
# Raises: TypeError: fun1() missing 1 required positional argument: 'arg3'
fun1(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun1() got an unexpected keyword argument 'arg4'
```

Pour les fonctions qui ont des arguments facultatifs, vous pouvez empaqueter les arguments sous forme de dictionnaire de la même manière:

```
fun2(**d)
# Returns: (1, 2, 3)
```

Mais là, vous pouvez omettre des valeurs, car elles seront remplacées par les valeurs par défaut:

```
fun2(**{'arg2': 2})
# Returns: ('a', 2, 'c')
```

Et comme précédemment, vous ne pouvez pas donner de valeurs supplémentaires qui ne sont pas des paramètres existants:

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun2() got an unexpected keyword argument 'arg4'
```

Dans l'utilisation du monde réel, les fonctions peuvent avoir des arguments à la fois positionnels et facultatifs, et cela fonctionne de la même façon:

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

vous pouvez appeler la fonction avec une simple itération:

```
fun3(*[1])
# Returns: (1, 'b', 'c')
fun3(*[1,2,3])
# Returns: (1, 2, 3)
```

ou avec juste un dictionnaire:

```
fun3(**{'arg1':1})
# Returns: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Returns: (1, 2, 3)
```

ou vous pouvez utiliser les deux dans le même appel:

```
fun3(*[1,2], **{'arg3':3})
# Returns: (1,2,3)
```

Attention cependant, vous ne pouvez pas fournir plusieurs valeurs pour le même argument:

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# Raises: TypeError: fun3() got multiple values for argument 'arg2'
```

Déballage des arguments de la fonction

Lorsque vous souhaitez créer une fonction pouvant accepter un nombre quelconque d'arguments, et ne pas appliquer la position ou le nom de l'argument à la compilation, il est possible et voici comment:

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

Les paramètres `*args` et `**kwargs` sont des paramètres spéciaux définis respectivement sur un `tuple` et un `dict` :

```
fun1(1,2,3)
# Prints: (1, 2, 3) {}
fun1(a=1, b=2, c=3)
# Prints: () {'a': 1, 'b': 2, 'c': 3}
fun1('x', 'y', 'z', a=1, b=2, c=3)
# Prints: ('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
```

Si vous regardez suffisamment de code Python, vous découvrirez rapidement qu'il est largement utilisé lors du passage d'arguments à une autre fonction. Par exemple, si vous souhaitez étendre la classe de chaîne:

```
class MyString(str):
    def __init__(self, *args, **kwarg):
        print('Constructing MyString')
        super(MyString, self).__init__(*args, **kwarg)
```

Lire Déstructurer la liste (aka emballage et déballage) en ligne:
https://riptutorial.com/fr/python/topic/4282/destructurer-la-liste--aka-emballage-et-deballage-

Chapitre 49: dictionnaire

Syntaxe

- mydict = {}
- mydict [k] = valeur
- valeur = mydict [k]
- valeur = mydict.get (k)
- value = mydict.get (k, "default_value")

Paramètres

Paramètre	Détails
clé	La clé à rechercher
valeur	La valeur à définir ou à renvoyer

Remarques

Éléments utiles à retenir lors de la création d'un dictionnaire:

- Chaque clé doit être **unique** (sinon elle sera remplacée)
- Chaque clé doit être **hashable** (peut utiliser le `hash` fonction de hachage il, sinon `TypeError` sera jeté)
- Il n'y a pas d'ordre particulier pour les clés.

Examples

Accéder aux valeurs d'un dictionnaire

```
dictionary = {"Hello": 1234, "World": 5678}
print(dictionary["Hello"])
```

Le code ci-dessus imprimera `1234` .

La chaîne `"Hello"` dans cet exemple est appelée une *clé* . Il est utilisé pour rechercher une valeur dans le `dict` en plaçant la clé entre crochets.

Le nombre `1234` est vu après les deux points respectifs dans la définition `dict` . Ceci s'appelle la *valeur à* laquelle `"Hello"` *correspond* dans ce `dict` .

Rechercher une valeur comme celle-ci avec une clé inexistante `KeyError` une exception `KeyError` , stoppant l'exécution si elle n'est pas `KeyError` . Si nous voulons accéder à une valeur sans risquer

une `KeyError` , nous pouvons utiliser la méthode `dictionary.get` . Par défaut, si la clé n'existe pas, la méthode renvoie `None` . Nous pouvons lui transmettre une seconde valeur à retourner au lieu de `None` en cas d'échec de la recherche.

```
w = dictionary.get("whatever")
x = dictionary.get("whatever", "nuh-uh")
```

Dans cet exemple, `w` obtiendra la valeur `None` et `x` obtiendra la valeur `"nuh-uh"` .

Le constructeur dict ()

Le constructeur `dict()` peut être utilisé pour créer des dictionnaires à partir d'arguments de mots-clés, ou à partir d'une seule itération de paires clé-valeur, ou à partir d'un seul dictionnaire et d'arguments de mots-clés.

```
dict(a=1, b=2, c=3)                   # {'a': 1, 'b': 2, 'c': 3}
dict([('d', 4), ('e', 5), ('f', 6)])  # {'d': 4, 'e': 5, 'f': 6}
dict([('a', 1)], b=2, c=3)            # {'a': 1, 'b': 2, 'c': 3}
dict({'a' : 1, 'b' : 2}, c=3)         # {'a': 1, 'b': 2, 'c': 3}
```

Éviter les exceptions de KeyError

Un écueil courant lors de l'utilisation des dictionnaires est d'accéder à une clé inexistante. Cela se traduit généralement par une exception `KeyError`

```
mydict = {}
mydict['not there']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

Un moyen d'éviter les erreurs de clé consiste à utiliser la méthode `dict.get` , qui vous permet de spécifier une valeur par défaut à renvoyer dans le cas d'une clé absente.

```
value = mydict.get(key, default_value)
```

Qui renvoie `mydict[key]` s'il existe, mais renvoie `default_value` . Notez que cela n'ajoute pas de `key` à `mydict` . Donc , si vous voulez conserver cette valeur paire de clés, vous devez utiliser `mydict.setdefault(key, default_value)` , qui *ne* stocke la paire de valeurs clés.

```
mydict = {}
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}
print(mydict.setdefault("foo", "bar"))
# bar
```

```
print(mydict)
# {'foo': 'bar'}
```

Une autre façon de faire face au problème est d'accepter l'exception

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

Vous pouvez également vérifier si la clé est `in` le dictionnaire.

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

Notez toutefois que, dans les environnements multithread, la clé peut être supprimée du dictionnaire après vérification, créant ainsi une condition de concurrence où l'exception peut toujours être levée.

Une autre option consiste à utiliser une sous-classe de dict, collections.defaultdict, qui possède une propriété default_factory pour créer de nouvelles entrées dans le dict lorsqu'une nouvelle clé est donnée.

Accéder aux clés et aux valeurs

Lorsque vous travaillez avec des dictionnaires, il est souvent nécessaire d'accéder à toutes les clés et valeurs du dictionnaire, soit dans une boucle `for` , une compréhension de liste, ou simplement sous la forme d'une liste simple.

Étant donné un dictionnaire comme:

```
mydict = {
    'a': '1',
    'b': '2'
}
```

Vous pouvez obtenir une liste de clés en utilisant la méthode `keys()` :

```
print(mydict.keys())
# Python2: ['a', 'b']
# Python3: dict_keys(['b', 'a'])
```

Si, au contraire, vous souhaitez une liste de valeurs, utilisez la méthode `values()` :

```
print(mydict.values())
# Python2: ['1', '2']
# Python3: dict_values(['2', '1'])
```

Si vous voulez travailler avec la clé et la valeur correspondante, vous pouvez utiliser la méthode `items()` :

```
print(mydict.items())
# Python2: [('a', '1'), ('b', '2')]
# Python3: dict_items([('b', '2'), ('a', '1')])
```

Remarque: comme un `dict` est non trié, `keys()` , `values()` et `items()` n'ont aucun ordre de tri. Utilisez `sort()` , `sorted()` , ou un `OrderedDict` si vous vous souciez de l'ordre que ces méthodes renvoient.

Différence Python 2/3: Dans Python 3, ces méthodes renvoient des objets itérables spéciaux, pas des listes, et sont l'équivalent des `iterkeys()` Python 2 `iterkeys()` , `itervalues()` et `iteritems()` . Ces objets peuvent être utilisés comme des listes pour la plupart, bien qu'il y ait des différences. Voir PEP 3106 pour plus de détails.

Introduction au dictionnaire

Un dictionnaire est un exemple de *magasin de valeurs de clé* également appelé *Mapping* in Python. Il vous permet de stocker et de récupérer des éléments en référençant une clé. Comme les dictionnaires sont référencés par clé, ils ont des recherches très rapides. Comme ils sont principalement utilisés pour référencer les éléments par clé, ils ne sont pas triés.

créer un dict

Les dictionnaires peuvent être initiés de plusieurs manières:

syntaxe littérale

```
d = {}                          # empty dict
d = {'key': 'value'}            # dict with initial values
```

Python 3.x 3.5

```
# Also unpacking one or multiple dictionaries with the literal syntax is possible

# makes a shallow copy of otherdict
d = {**otherdict}
# also updates the shallow copy with the contents of the yetanotherdict.
d = {**otherdict, **yetanotherdict}
```

dict compréhension

```
d = {k:v for k,v in [('key', 'value',)]}
```

voir aussi: Compréhensions

classe intégrée: `dict()`

```
d = dict()                      # emtpy dict
d = dict(key='value')           # explicit keyword arguments
d = dict([('key', 'value')])  # passing in a list of key/value pairs
# make a shallow copy of another dict (only possible if keys are only strings!)
d = dict(**otherdict)
```

modifier un dict

Pour ajouter des éléments à un dictionnaire, créez simplement une nouvelle clé avec une valeur:

```
d['newkey'] = 42
```

Il est également possible d'ajouter une `list` et un `dictionary` tant que valeur:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

Pour supprimer un élément, supprimez la clé du dictionnaire:

```
del d['newkey']
```

Dictionnaire avec les valeurs par défaut

Disponible dans la bibliothèque standard en tant que `defaultdict`

```
from collections import defaultdict

d = defaultdict(int)
d['key']                          # 0
d['key'] = 5
d['key']                          # 5

d = defaultdict(lambda: 'empty')
d['key']                          # 'empty'
d['key'] = 'full'
d['key']                          # 'full'
```

[*] Alternativement, si vous devez utiliser la classe `dict` using `dict.setdefault()` , l' `using dict.setdefault()` vous permettra de créer une valeur par défaut à chaque fois que vous accédez à une clé qui n'existait pas auparavant:

```
>>> d = {}
{}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
{'Another_key': ['This worked!']}
```

Gardez à l'esprit que si vous avez beaucoup de valeurs à ajouter, `dict.setdefault()` créera une nouvelle instance de la valeur initiale (dans cet exemple, a `[]`) à chaque appel, ce qui peut créer des charges de travail inutiles.

[*] *Python Cookbook, 3ème édition, par David Beazley et Brian K. Jones (O'Reilly). Copyright 2013 David Beazley et Brian Jones, 978-1-449-34037-7.*

Créer un dictionnaire ordonné

Vous pouvez créer un dictionnaire ordonné qui suivra un ordre déterminé lors d'une itération sur les clés du dictionnaire.

Utilisez `OrderedDict` du module de `collections` . Cela renverra toujours les éléments du dictionnaire dans l'ordre d'insertion d'origine une fois l'itération terminée.

```
from collections import OrderedDict

d = OrderedDict()
d['first'] = 1
d['second'] = 2
d['third'] = 3
d['last'] = 4

# Outputs "first 1", "second 2", "third 3", "last 4"
for key in d:
    print(key, d[key])
```

Déballage des dictionnaires à l'aide de l'opérateur **

Vous pouvez utiliser l'opérateur de décompression d'argument de mot clé `**` pour fournir les paires clé-valeur dans un dictionnaire aux arguments d'une fonction. Un exemple simplifié de la documentation officielle :

```
>>>
>>> def parrot(voltage, state, action):
...     print("This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)

This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

A partir de Python 3.5, vous pouvez également utiliser cette syntaxe pour fusionner un nombre arbitraire d'objets `dict` .

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
>>> fishdog = {**fish, **dog}
>>> fishdog

{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Comme le montre cet exemple, les clés en double correspondent à leur valeur la plus récente (par exemple, "Clifford" remplace "Nemo").

Fusion de dictionnaires

Considérons les dictionnaires suivants:

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
```

Python 3.5+

```
>>> fishdog = {**fish, **dog}
>>> fishdog
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Comme le montre cet exemple, les clés en double correspondent à leur valeur la plus récente (par exemple, "Clifford" remplace "Nemo").

Python 3.3+

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

Avec cette technique, la valeur la plus importante prime sur une clé donnée plutôt que sur la dernière ("Clifford" est rejeté en faveur de "Nemo").

Python 2.x, 3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Cela utilise la dernière valeur, comme avec la technique basée sur `**` pour la fusion ("Clifford" remplace "Nemo").

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update` utilise le dernier dict pour écraser le précédent.

La virgule de fin

Comme les listes et les tuples, vous pouvez inclure une virgule de fin dans votre dictionnaire.

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

PEP 8 stipule que vous devez laisser un espace entre la virgule finale et l'accolade de fermeture.

Toutes les combinaisons de valeurs de dictionnaire

```
options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}
```

Étant donné un dictionnaire tel que celui montré ci-dessus, où il y a une liste représentant un ensemble de valeurs à explorer pour la clé correspondante. Supposons que vous vouliez explorer `"x"="a"` avec `"y"=10` , puis `"x"="a"` avec `"y"=10` , et ainsi de suite jusqu'à ce que vous ayez exploré toutes les combinaisons possibles.

Vous pouvez créer une liste qui renvoie toutes ces combinaisons de valeurs en utilisant le code suivant.

```
import itertools

options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]}

keys = options.keys()
values = (options[key] for key in keys)
combinations = [dict(zip(keys, combination)) for combination in itertools.product(*values)]
print combinations
```

Cela nous donne la liste suivante stockée dans les `combinations` variables:

```
[{'x': 'a', 'y': 10},
 {'x': 'b', 'y': 10},
 {'x': 'a', 'y': 20},
 {'x': 'b', 'y': 20},
 {'x': 'a', 'y': 30},
 {'x': 'b', 'y': 30}]
```

Itérer sur un dictionnaire

Si vous utilisez un dictionnaire en tant qu'itérateur (par exemple dans une déclaration `for`), il parcourt les **clés** du dictionnaire. Par exemple:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
```

```
    print(key, d[key])
# c 3
# b 2
# a 1
```

La même chose est vraie lorsqu'elle est utilisée dans une compréhension

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x 3.0

La méthode `items()` peut être utilisée pour parcourir la **clé** et la **valeur** simultanément:

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

Alors que la méthode `values()` peut être utilisée pour itérer uniquement les valeurs, comme prévu:

```
for key, value in d.values():
    print(key, value)
    # 3
    # 2
    # 1
```

Python 2.x 2.2

Ici, les méthodes `keys()` , `values()` et `items()` renvoient des listes, et il y a les trois méthodes supplémentaires `iterkeys()` `itervalues()` et `iteritems()` pour renvoyer les iteraters.

Créer un dictionnaire

Règles pour créer un dictionnaire:

- Chaque clé doit être **unique** (sinon elle sera remplacée)
- Chaque clé doit être **hashable** (peut utiliser le `hash` fonction de hachage il, sinon `TypeError` sera jeté)
- Il n'y a pas d'ordre particulier pour les clés.

```
# Creating and populating it with values
stock = {'eggs': 5, 'milk': 2}

# Or creating an empty dictionary
dictionary = {}

# And populating it after
dictionary['eggs'] = 5
dictionary['milk'] = 2

# Values can also be lists
```

```
mydict = {'a': [1, 2, 3], 'b': ['one', 'two', 'three']}

# Use list.append() method to add new elements to the values list
mydict['a'].append(4)    # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three']}
mydict['b'].append('four')  # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three', 'four']}

# We can also create a dictionary using a list of two-items tuples
iterable = [('eggs', 5), ('milk', 2)]
dictionary = dict(iterables)

# Or using keyword argument:
dictionary = dict(eggs=5, milk=2)

# Another way will be to use the dict.fromkeys:
dictionary = dict.fromkeys((milk, eggs))  # => {'milk': None, 'eggs': None}
dictionary = dict.fromkeys((milk, eggs), (2, 5))  # => {'milk': 2, 'eggs': 5}
```

Exemple de dictionnaires

Dictionnaires mappent les clés aux valeurs.

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

Les valeurs du dictionnaire sont accessibles par leurs clés.

```
print "Little " + car["color"] + " " + car["model"] + "!"
# This would print out "Little Red Corvette!"
```

Les dictionnaires peuvent également être créés dans un style JSON:

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

Les valeurs de dictionnaire peuvent être réitérées sur:

```
for key in car:
  print key + ": " + car[key]

# wheels: 4
# color: Red
# model: Corvette
```

Lire dictionnaire en ligne: https://riptutorial.com/fr/python/topic/396/dictionnaire

Chapitre 50: Différence entre module et package

Remarques

Il est possible de mettre un paquet Python dans un fichier ZIP et de l'utiliser de cette manière si vous ajoutez ces lignes au début de votre script:

```
import sys
sys.path.append("package.zip")
```

Examples

Modules

Un module est un fichier Python unique pouvant être importé. Utiliser un module ressemble à ceci:

`module.py`

```
def hi():
    print("Hello world!")
```

`my_script.py`

```
import module
module.hi()
```

dans un interprète

```
>>> from module import hi
>>> hi()
# Hello world!
```

Paquets

Un paquet est composé de plusieurs fichiers Python (ou modules) et peut même inclure des bibliothèques écrites en C ou C ++. Au lieu d'être un fichier unique, il s'agit d'une structure de dossiers complète qui pourrait ressembler à ceci:

`package` dossier

- `__init__.py`
- `dog.py`
- `hi.py`

`__init__.py`

```
from package.dog import woof
from package.hi import hi
```

`dog.py`

```
def woof():
    print("WOOF!!!")
```

`hi.py`

```
def hi():
    print("Hello world!")
```

Tous les packages Python doivent contenir un fichier `__init__.py` . Lorsque vous importez un package dans votre script (`import package`), le script `__init__.py` sera exécuté, vous donnant accès à toutes les fonctions du package. Dans ce cas, il vous permet d'utiliser les fonctions `package.hi` et `package.woof` .

Lire Différence entre module et package en ligne:
https://riptutorial.com/fr/python/topic/3142/difference-entre-module-et-package

Chapitre 51: Distribution

Examples

py2app

Pour utiliser le framework py2app, vous devez d'abord l'installer. Pour ce faire, ouvrez le terminal et entrez la commande suivante:

```
sudo easy_install -U py2app
```

Vous pouvez également installer les paquetages en `pip` :

```
pip install py2app
```

Ensuite, créez le fichier d'installation pour votre script python:

```
py2applet --make-setup MyApplication.py
```

Modifiez les paramètres du fichier de configuration à votre convenance, il s'agit de la valeur par défaut:

```
"""
This is a setup.py script generated by py2applet

Usage:
    python setup.py py2app
"""

from setuptools import setup

APP = ['test.py']
DATA_FILES = []
OPTIONS = {'argv_emulation': True}

setup(
    app=APP,
    data_files=DATA_FILES,
    options={'py2app': OPTIONS},
    setup_requires=['py2app'],
)
```

Pour ajouter un fichier icône (ce fichier doit avoir une extension .icns) ou inclure des images dans votre application comme référence, modifiez vos options comme indiqué:

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

Enfin, entrez ceci dans le terminal:

```
python setup.py py2app
```

Le script devrait s'exécuter et vous trouverez votre application terminée dans le dossier dist.

Utilisez les options suivantes pour plus de personnalisation:

```
optimize (-O)          optimization level: -O1 for "python -O", -O2 for
                       "python -OO", and -O0 to disable [default: -O0]

includes (-i)          comma-separated list of modules to include

packages (-p)          comma-separated list of packages to include

extension              Bundle extension [default:.app for app, .plugin for
                       plugin]

extra-scripts          comma-separated list of additional scripts to include
                       in an application or plugin.
```

cx_Freeze

Installez cx_Freeze d' ici

Décompressez le dossier et exécutez ces commandes à partir de ce répertoire:

```
python setup.py build
sudo python setup.py install
```

Créez un nouveau répertoire pour votre script python et créez un fichier **"setup.py"** dans le même répertoire avec le contenu suivant:

```
application_title = "My Application" # Use your own application name
main_python_file = "my_script.py" # Your python script

import sys

from cx_Freeze import setup, Executable

base = None
if sys.platform == "win32":
    base = "Win32GUI"

includes = ["atexit","re"]

setup(
        name = application_title,
        version = "0.1",
        description = "Your Description",
        options = {"build_exe" : {"includes" : includes }},
        executables = [Executable(main_python_file, base = base)])
```

Exécutez maintenant votre setup.py à partir du terminal:

```
python setup.py bdist_mac
```

NOTE: Sur El Capitan, il faudra que ce soit exécuté en tant que root avec le mode SIP désactivé.

Lire Distribution en ligne: https://riptutorial.com/fr/python/topic/2026/distribution

Chapitre 52: Django

Introduction

Django est un framework Web Python de haut niveau qui encourage un développement rapide et une conception propre et pragmatique. Conçu par des développeurs expérimentés, il prend en charge une grande partie des problèmes de développement Web, vous pouvez donc vous concentrer sur l'écriture de votre application sans avoir à réinventer la roue. C'est gratuit et open source.

Examples

Bonjour tout le monde avec Django

Créez un exemple simple de `Hello World` utilisant votre django.

veillons à ce que django soit d'abord installé sur votre PC.

ouvrir un terminal et taper: python -c "import django"
-> si aucune erreur ne survient, cela signifie que Django est déjà installé.

Maintenant, créons un projet dans django. Pour cela écrivez ci-dessous la commande sur le terminal:
django-admin startproject HelloWorld

La commande ci-dessus créera un répertoire nommé HelloWorld.
La structure du répertoire sera comme suit:
Bonjour le monde
| --helloworld
| | - **init** .py
| |settings.py
| | --urls.py
| | --wsgi.py
| --manage.py

Écriture de vues (Référence à partir de la documentation de Django)

Une fonction de vue, ou vue en abrégé, est simplement une fonction Python qui prend une requête Web et renvoie une réponse Web. Cette réponse peut être le contenu HTML d'une page Web ou autre. La documentation dit que nous pouvons écrire des vues dans n'importe quelle fonction, mais il est préférable d'écrire dans views.py placé dans notre répertoire de projet.

Voici une vue qui renvoie un message hello world (views.py)

```
from django.http import HttpResponse
```

```
define helloWorld(request):
    return HttpResponse("Hello World!! Django Welcomes You.")
```

comprenons le code, pas à pas.

- Tout d'abord, nous importons la classe HttpResponse du module django.http.
- Ensuite, nous définissons une fonction appelée helloWorld. Ceci est la fonction d'affichage. Chaque fonction de vue prend un objet HttpRequest comme premier paramètre, qui est généralement appelé request.

  Notez que le nom de la fonction d'affichage n'a pas d'importance; il n'a pas besoin d'être nommé d'une certaine manière pour que Django le reconnaisse. nous l'avons appelé helloWorld ici, de sorte que cela sera clair.
- La vue renvoie un objet HttpResponse qui contient la réponse générée. Chaque fonction de vue est chargée de retourner un objet HttpResponse.

Pour plus d'informations sur les vues de Django, cliquez ici

Mapper des URL à des vues
Pour afficher cette vue sur une URL particulière, vous devez créer un URLconf;

Avant cela, comprenons comment django traite les requêtes.

- Django détermine le module URLconf racine à utiliser.
- Django charge ce module Python et recherche la variable urlpatterns. Cela devrait être une liste Python d'instances django.conf.urls.url ().
- Django parcourt chaque modèle d'URL, dans l'ordre, et s'arrête au premier correspondant à l'URL demandée.
- Une fois que l'une des expressions régulières correspond, Django importe et appelle la vue donnée, qui est une simple fonction Python.

Voici comment notre URLconf se ressemble:

```
from django.conf.urls import url
from . import views #import the views.py from current directory

urlpatterns = [
   url(r'^helloworld/$', views.helloWorld),
]
```

Pour plus d'informations sur les Urs de Django, cliquez ici

Changez maintenant le répertoire en HelloWorld et écrivez ci-dessous la commande sur le terminal.
python manage.py runserver

par défaut, le serveur sera exécuté à 127.0.0.1:8000

Ouvrez votre navigateur et tapez 127.0.0.1:8000/helloworld/. La page vous montrera "Bonjour tout

le monde! Django vous souhaite la bienvenue".

Lire Django en ligne: https://riptutorial.com/fr/python/topic/8994/django

Chapitre 53: Données binaires

Syntaxe

- pack (fmt, v1, v2, ...)
- décompresser (fmt, buffer)

Examples

Mettre en forme une liste de valeurs dans un objet octet

```
from struct import pack

print(pack('I3c', 123, b'a', b'b', b'c'))  # b'{\x00\x00\x00abc'
```

Décompresser un objet octet selon une chaîne de format

```
from struct import unpack

print(unpack('I3c', b'{\x00\x00\x00abc'))  # (123, b'a', b'b', b'c')
```

Emballage d'une structure

Le module " **struct** " permet de conditionner des objets python en tant que bloc d'octets contigu ou de dissocier un bloc d'octets en structures python.

La fonction pack prend une chaîne de format et un ou plusieurs arguments et renvoie une chaîne binaire. Cela ressemble beaucoup à ce que vous formiez une chaîne, sauf que la sortie n'est pas une chaîne mais un bloc d'octets.

```
import struct
import sys
print "Native byteorder: ", sys.byteorder
# If no byteorder is specified, native byteorder is used
buffer = struct.pack("ihb", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
print "Byte chunk unpacked: ", struct.unpack("ihb", buffer)
# Last element as unsigned short instead of unsigned char ( 2 Bytes)
buffer = struct.pack("ihh", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
```

Sortie:

> Ordre d'exécution natif: petit bloc d'octet: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x00 \ x05' Bloc d'octets décompressé: (3, 4, 5) Bloc d'octets: '\ x03 \ x00 \ x00 \ x00 \ x04 \ x00 \ x05 \ x00 '

Vous pouvez utiliser l'ordre des octets du réseau avec les données reçues du réseau ou des données de pack pour les envoyer au réseau.

```
import struct
# If no byteorder is specified, native byteorder is used
buffer = struct.pack("hhh", 3, 4, 5)
print "Byte chunk native byte order: ", repr(buffer)
buffer = struct.pack("!hhh", 3, 4, 5)
print "Byte chunk network byte order: ", repr(buffer)
```

Sortie:

> Ordre d'octet natif de bloc d'octets: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00'
>
> Ordre des octets du réseau de blocs d'octets: '\ x00 \ x03 \ x00 \ x04 \ x00 \ x05'

Vous pouvez optimiser en évitant la surcharge liée à l'allocation d'un nouveau tampon en fournissant un tampon créé précédemment.

```
import struct
from ctypes import create_string_buffer
bufferVar = create_string_buffer(8)
bufferVar2 = create_string_buffer(8)
# We use a buffer that has already been created
# provide format, buffer, offset and data
struct.pack_into("hhh", bufferVar, 0, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar.raw)
struct.pack_into("hhh", bufferVar2, 2, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar2.raw)
```

Sortie:

> Bloc d'octets: '\ x03 \ x00 \ x04 \ x00 \ x05 \ x00 \ x00 \ x00'
>
> Bloc d'octets: '\ x00 \ x00 \ x03 \ x00 \ x04 \ x00 \ x05 \ x00'

Lire Données binaires en ligne: https://riptutorial.com/fr/python/topic/2978/donnees-binaires

Chapitre 54: Douilles

Introduction

De nombreux langages de programmation utilisent des sockets pour communiquer entre les processus ou entre les périphériques. Cette rubrique explique l'utilisation appropriée du module sockets dans Python pour faciliter l'envoi et la réception de données via les protocoles réseau courants.

Paramètres

Paramètre	La description
socket.AF_UNIX	Socket UNIX
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

Examples

Envoi de données via UDP

UDP est un protocole sans connexion. Les messages vers d'autres processus ou ordinateurs sont envoyés sans établir de connexion. Il n'y a pas de confirmation automatique si votre message a été reçu. UDP est généralement utilisé dans les applications sensibles à la latence ou dans les applications qui envoient des diffusions à l'échelle du réseau.

Le code suivant envoie un message à un processus écoutant sur le port localhost 6667 en utilisant UDP

Notez qu'il n'est pas nécessaire de "fermer" le socket après l'envoi, car UDP est sans connexion .

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
msg = ("Hello you there!").encode('utf-8')  # socket.sendto() takes bytes as input, hence we
must encode the string first.
s.sendto(msg, ('localhost', 6667))
```

Recevoir des données via UDP

UDP est un protocole sans connexion. Cela signifie que les pairs qui envoient des messages ne nécessitent pas d'établir de connexion avant d'envoyer des messages. `socket.recvfrom` retourne donc un tuple (`msg` [le message reçu par le socket], `addr` [l'adresse de l'expéditeur])

Un serveur UDP utilisant uniquement le module `socket` :

```
from socket import socket, AF_INET, SOCK_DGRAM
sock = socket(AF_INET, SOCK_DGRAM)
sock.bind(('localhost', 6667))

while True:
    msg, addr = sock.recvfrom(8192)  # This is the amount of bytes to read at maximum
    print("Got message from %s: %s" % (addr, msg))
```

Voici une implémentation alternative utilisant `socketserver.UDPServer` :

```
from socketserver import BaseRequestHandler, UDPServer

class MyHandler(BaseRequestHandler):
    def handle(self):
        print("Got connection from: %s" % self.client_address)
        msg, sock = self.request
        print("It said: %s" % msg)
        sock.sendto("Got your message!".encode(), self.client_address) # Send reply

serv = UDPServer(('localhost', 6667), MyHandler)
serv.serve_forever()
```

Par défaut, bloc de `sockets` . Cela signifie que l'exécution du script attendra que le socket reçoive des données.

Envoi de données via TCP

L'envoi de données via Internet est possible grâce à plusieurs modules. Le module sockets fournit un accès de bas niveau aux opérations du système d'exploitation sous-jacentes responsables de l'envoi ou de la réception de données provenant d'autres ordinateurs ou processus.

Le code suivant envoie la chaîne d'octets `b'Hello'` à un serveur TCP écoutant sur le port 6667 sur l'hôte localhost et ferme la connexion lorsque vous `b'Hello'` terminé:

```
from socket import socket, AF_INET, SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 6667))  # The address of the TCP server listening
s.send(b'Hello')
s.close()
```

La sortie Socket est bloquée par défaut, ce qui signifie que le programme attendra la connexion et enverra des appels jusqu'à ce que l'action soit terminée. Pour la connexion, cela signifie que le serveur accepte réellement la connexion. Pour l'envoi, cela signifie uniquement que le système d'exploitation dispose de suffisamment d'espace tampon pour mettre en file d'attente les données à envoyer ultérieurement.

Les prises doivent toujours être fermées après utilisation.

Serveur TCP multi-thread

Lorsqu'il est exécuté sans arguments, ce programme démarre un serveur socket TCP qui écoute les connexions à `127.0.0.1` sur le port `5000` . Le serveur gère chaque connexion dans un thread séparé.

Lorsqu'il est exécuté avec l'argument `-c` , ce programme se connecte au serveur, lit la liste des clients et l'imprime. La liste de clients est transférée sous forme de chaîne JSON. Le nom du client peut être spécifié en passant l'argument `-n` . En passant des noms différents, l'effet sur la liste des clients peut être observé.

client_list.py

```
import argparse
import json
import socket
import threading

def handle_client(client_list, conn, address):
    name = conn.recv(1024)
    entry = dict(zip(['name', 'address', 'port'], [name, address[0], address[1]]))
    client_list[name] = entry
    conn.sendall(json.dumps(client_list))
    conn.shutdown(socket.SHUT_RDWR)
    conn.close()

def server(client_list):
    print "Starting server..."
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind(('127.0.0.1', 5000))
    s.listen(5)
    while True:
        (conn, address) = s.accept()
        t = threading.Thread(target=handle_client, args=(client_list, conn, address))
        t.daemon = True
        t.start()

def client(name):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('127.0.0.1', 5000))
    s.send(name)
    data = s.recv(1024)
    result = json.loads(data)
    print json.dumps(result, indent=4)

def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', dest='client', action='store_true')
    parser.add_argument('-n', dest='name', type=str, default='name')
    result = parser.parse_args()
    return result

def main():
    client_list = dict()
```

```
    args = parse_arguments()
    if args.client:
        client(args.name)
    else:
        try:
            server(client_list)
        except KeyboardInterrupt:
            print "Keyboard interrupt"

if __name__ == '__main__':
    main()
```

Sortie du serveur

```
$ python client_list.py
Starting server...
```

Sortie client

```
$ python client_list.py -c -n name1
{
    "name1": {
        "address": "127.0.0.1",
        "port": 62210,
        "name": "name1"
    }
}
```

Les tampons de réception sont limités à 1024 octets. Si la représentation de la chaîne JSON de la liste de clients dépasse cette taille, elle sera tronquée. Cela provoquera la levée de l'exception suivante:

```
ValueError: Unterminated string starting at: line 1 column 1023 (char 1022)
```

Sockets Raw sous Linux

D'abord, vous désactivez la somme de contrôle automatique de votre carte réseau:

```
sudo ethtool -K eth1 tx off
```

Ensuite, envoyez votre paquet en utilisant un socket SOCK_RAW:

```
#!/usr/bin/env python
from socket import socket, AF_PACKET, SOCK_RAW
s = socket(AF_PACKET, SOCK_RAW)
s.bind(("eth1", 0))

# We're putting together an ethernet frame here,
# but you could have anything you want instead
# Have a look at the 'struct' module for more
# flexible packing/unpacking of binary data
# and 'binascii' for 32 bit CRC
src_addr = "\x01\x02\x03\x04\x05\x06"
```

```
dst_addr = "\x01\x02\x03\x04\x05\x06"
payload = ("["*30)+"PAYLOAD"+("]"*30)
checksum = "\x1a\x2b\x3c\x4d"
ethertype = "\x08\x01"

s.send(dst_addr+src_addr+ethertype+payload+checksum)
```

Lire Douilles en ligne: https://riptutorial.com/fr/python/topic/1530/douilles

Chapitre 55: Échancrure

Examples

Erreurs d'indentation

L'espacement doit être uniforme et uniforme partout. Une indentation incorrecte peut provoquer une `IndentationError` ou provoquer des imprévus dans le programme. L'exemple suivant déclenche une `IndentationError` :

```
a = 7
if a > 5:
  print "foo"
else:
  print "bar"
 print "done"
```

Ou si la ligne qui suit un deux-points n'est pas en retrait, une `IndentationError` d' `IndentationError` sera également déclenchée:

```
if True:
print "true"
```

Si vous ajoutez une indentation là où elle n'appartient pas, une `IndentationError` d' `IndentationError` sera `IndentationError` :

```
if  True:
    a = 6
        b = 5
```

Si vous oubliez de ne pas indenter la fonctionnalité pourrait être perdu. Dans cet exemple, `None` est renvoyé au lieu de `False` attendu:

```
def isEven(a):
    if a%2 ==0:
        return True
        #this next line should be even with the if
        return False
print isEven(7)
```

Exemple simple

Pour Python, Guido van Rossum a basé le regroupement des déclarations sur l'indentation. Les raisons en sont expliquées dans la première section de la "FAQ Python Design and History" . Colons, `:` , sont utilisés pour déclarer un bloc de code en retrait , comme dans l'exemple suivant:

```
class ExampleClass:
    #Every function belonging to a class must be indented equally
```

```
    def __init__(self):
        name = "example"

    def someFunction(self, a):
        #Notice everything belonging to a function must be indented
        if a > 5:
            return True
        else:
            return False

#If a function is not indented to the same level it will not be considers as part of the
parent class
def separateFunction(b):
    for i in b:
    #Loops are also indented and nested conditions start a new indentation
        if i == 1:
            return True
    return False

separateFunction([2,3,5,6,1])
```

Espaces ou onglets?

L' indentation recommandée est de 4 espaces, mais des tabulations ou des espaces peuvent être utilisés tant qu'ils sont cohérents. ***Ne mélangez pas les onglets et les espaces dans Python,*** car cela provoquerait une erreur dans Python 3 et pourrait provoquer des erreurs dans Python 2 .

Comment l'indentation est analysée

Les espaces blancs sont traités par l'analyseur lexical avant d'être analysés.

L'analyseur lexical utilise une pile pour stocker les niveaux d'indentation. Au début, la pile contient juste la valeur 0, qui est la position la plus à gauche. Chaque fois qu'un bloc imbriqué commence, le nouveau niveau d'indentation est poussé sur la pile et un jeton "INDENT" est inséré dans le flux de jetons qui est transmis à l'analyseur. Il ne peut jamais y avoir plus d'un jeton "INDENT" dans une ligne (`IndentationError`).

Lorsqu'une ligne est rencontrée avec un niveau d'indentation plus petit, les valeurs sont extraites de la pile jusqu'à ce que la valeur soit supérieure au nouveau niveau d'indentation (si aucune valeur n'est trouvée, une erreur de syntaxe se produit). Pour chaque valeur sautée, un jeton "DEDENT" est généré. De toute évidence, il peut y avoir plusieurs jetons "DEDENT" à la suite.

L'analyseur lexical ignore les lignes vides (celles ne contenant que des espaces et éventuellement des commentaires) et ne générera jamais de jeton "INDENT" ou "DEDENT" pour ces dernières.

A la fin du code source, des jetons "DEDENT" sont générés pour chaque niveau d'indentation laissé sur la pile, jusqu'à ce qu'il ne reste que le 0.

Par exemple:

```
if foo:
    if bar:
```

```
        x = 42
else:
    print foo
```

est analysé comme:

```
<if> <foo> <:>                    [0]
<INDENT> <if> <bar> <:>           [0, 4]
<INDENT> <x> <=> <42>             [0, 4, 8]
<DEDENT> <DEDENT> <else> <:>      [0]
<INDENT> <print> <foo>            [0, 2]
<DEDENT>
```

L'analyseur traite les jetons "INDENT" et "DEDENT" en tant que délimiteurs de bloc.

Lire Échancrure en ligne: https://riptutorial.com/fr/python/topic/2597/echancrure

Chapitre 56: Écrire dans un fichier CSV à partir d'une chaîne ou d'une liste

Introduction

L'écriture dans un fichier .csv n'est pas sans rappeler l'écriture dans un fichier régulier à tous égards, et est assez simple. Au mieux de mes capacités, je couvrirai l'approche la plus simple et la plus efficace du problème.

Paramètres

Paramètre	Détails
open (**"/ path /"** , "mode")	Spécifiez le chemin d'accès à votre fichier CSV
ouvert (chemin, **"mode"**)	Spécifiez le mode pour ouvrir le fichier (lecture, écriture, etc.)
csv.writer (**fichier** , délimiteur)	Passer le fichier CSV ouvert ici
csv.writer (fichier, **délimiteur = ''**)	Spécifier le caractère ou le motif du délimiteur

Remarques

```
open( path, "wb")
```

`"wb"` - Mode d'écriture.

Le paramètre `b` dans `"wb"` nous avons utilisé n'est nécessaire que si vous voulez l'ouvrir en mode binaire, ce qui n'est nécessaire que dans certains systèmes d'exploitation comme Windows.

```
csv.writer ( csv_file, delimiter=',' )
```

Ici, le délimiteur que nous avons utilisé est `,` car nous voulons que chaque cellule de données dans une rangée contienne respectivement le prénom, le nom de famille et l'âge. Étant donné que notre liste est divisée le long de la `,` aussi, il se révèle assez pratique pour nous.

Examples

Exemple d'écriture de base

```
import csv
```

```
#------ We will write to CSV in this function ------------

def csv_writer(data, path):

    #Open CSV file whose path we passed.
    with open(path, "wb") as csv_file:

        writer = csv.writer(csv_file, delimiter=',')
        for line in data:
            writer.writerow(line)



#---- Define our list here, and call function ------------

if __name__ == "__main__":

    """
    data = our list that we want to write.
    Split it so we get a list of lists.
    """
    data = ["first_name,last_name,age".split(","),
            "John,Doe,22".split(","),
            "Jane,Doe,31".split(","),
            "Jack,Reacher,27".split(",")
            ]

    # Path to CSV file we want to write to.
    path = "output.csv"
    csv_writer(data, path)
```

Ajout d'une chaîne en tant que nouvelle ligne dans un fichier CSV

```
def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "\n")
```

Lire Écrire dans un fichier CSV à partir d'une chaîne ou d'une liste en ligne: https://riptutorial.com/fr/python/topic/10862/ecrire-dans-un-fichier-csv-a-partir-d-une-chaine-ou-d-une-liste

Chapitre 57: Écrire des extensions

Examples

Bonjour tout le monde avec l'extension C

Le fichier source C suivant (que nous appellerons `hello.c` à des fins de démonstration) produit un module d'extension nommé `hello` qui contient une seule fonction `greet()` :

```
#include <Python.h>
#include <stdio.h>

#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif

static PyObject *hello_greet(PyObject *self, PyObject *args)
{
    const char *input;
    if (!PyArg_ParseTuple(args, "s", &input)) {
        return NULL;
    }
    printf("%s", input);
    Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] = {
    { "greet", hello_greet, METH_VARARGS, "Greet the user" },
    { NULL, NULL, 0, NULL }
};

#ifdef IS_PY3K
static struct PyModuleDef hellomodule = {
    PyModuleDef_HEAD_INIT, "hello", NULL, -1, HelloMethods
};

PyMODINIT_FUNC PyInit_hello(void)
{
    return PyModule_Create(&hellomodule);
}
#else
PyMODINIT_FUNC inithello(void)
{
    (void) Py_InitModule("hello", HelloMethods);
}
#endif
```

Pour compiler le fichier avec le compilateur `gcc` , exécutez la commande suivante dans votre terminal préféré:

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

Pour exécuter la fonction `greet()` que nous avons écrite précédemment, créez un fichier dans le même répertoire et appelez-le `hello.py`

```
import hello          # imports the compiled library
hello.greet("Hello!") # runs the greet() function with "Hello!" as an argument
```

Passer un fichier ouvert à C Extensions

Transmettez un objet fichier ouvert de Python au code d'extension C.

Vous pouvez convertir le fichier en un descripteur de fichier entier à l'aide de la fonction `PyObject_AsFileDescriptor` :

```
PyObject *fobj;
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0){
    return NULL;
}
```

Pour reconvertir un descripteur de fichier entier en objet python, utilisez `PyFile_FromFd` .

```
int fd; /* Existing file descriptor */
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

Extension C utilisant c ++ et Boost

Ceci est un exemple de base d'une *extension* C utilisant C ++ et Boost .

Code C ++

Code C ++ placé dans hello.cpp:

```
#include <boost/python/module.hpp>
#include <boost/python/list.hpp>
#include <boost/python/class.hpp>
#include <boost/python/def.hpp>

// Return a hello world string.
std::string get_hello_function()
{
   return "Hello world!";
}

// hello class that can return a list of count hello world strings.
class hello_class
{
public:

   // Taking the greeting message in the constructor.
   hello_class(std::string message) : _message(message) {}

   // Returns the message count times in a python list.
   boost::python::list as_list(int count)
   {
      boost::python::list res;
```

```
      for (int i = 0; i < count; ++i) {
         res.append(_message);
      }
      return res;
   }

private:
   std::string _message;
};


// Defining a python module naming it to "hello".
BOOST_PYTHON_MODULE(hello)
{
   // Here you declare what functions and classes that should be exposed on the module.

   // The get_hello_function exposed to python as a function.
   boost::python::def("get_hello", get_hello_function);

   // The hello_class exposed to python as a class.
   boost::python::class_<hello_class>("Hello", boost::python::init<std::string>())
      .def("as_list", &hello_class::as_list)
      ;
}
```

Pour compiler cela dans un module python, vous aurez besoin des en-têtes python et des bibliothèques boost. Cet exemple a été réalisé sur Ubuntu 12.04 en utilisant python 3.4 et gcc. Boost est pris en charge sur de nombreuses plates-formes. Dans le cas d'Ubuntu, les paquets nécessaires ont été installés en utilisant:

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

Compiler le fichier source dans un fichier .so pouvant être importé ultérieurement en tant que module, à condition que ce soit sur le chemin Python:

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -
lboost_system -l:libpython3.4m.so
```

Le code python dans le fichier example.py:

```
import hello

print(hello.get_hello())

h = hello.Hello("World hello!")
print(h.as_list(3))
```

Ensuite, `python3 example.py` donnera la sortie suivante:

```
Hello world!
['World hello!', 'World hello!', 'World hello!']
```

Lire Écrire des extensions en ligne: https://riptutorial.com/fr/python/topic/557/ecrire-des-extensions

Chapitre 58: Empiler

Introduction

Une pile est un conteneur d'objets insérés et retirés selon le principe du dernier entré, premier sorti (LIFO). Dans les piles pushdown, seules deux opérations sont autorisées: **enfoncez l'élément dans la pile et sortez-le de la pile** . Une pile est une structure de données à accès limité - des **éléments peuvent être ajoutés et supprimés de la pile uniquement en haut** . Voici une définition structurelle d'une pile: une pile est soit vide, soit constituée d'un sommet et le reste d'une pile.

Syntaxe

- stack = [] # Crée la pile
- stack.append (object) # Ajoute un objet au sommet de la pile
- stack.pop () -> object # Renvoie le plus haut objet de la pile et le supprime également
- list [-1] -> object # Jetez un coup d'oeil sur l'objet le plus haut sans le supprimer

Remarques

De Wikipedia :

> En informatique, une *pile* est un type de données abstrait qui sert de collection d'éléments, avec deux opérations principales: *push* , qui ajoute un élément à la collection, et *pop* , qui supprime l'élément le plus récemment ajouté qui n'a pas encore été supprimé.

En raison de la façon dont leurs éléments sont accessibles, les piles sont également connus comme *Last In, First Out (LIFO) piles.*

En Python, on peut utiliser des listes comme piles avec `append()` comme push et `pop()` comme opérations pop. Les deux opérations sont exécutées à temps constant O (1).

La structure de données `deque` de Python peut également être utilisée comme une pile. Par rapport aux listes, les `deque` permettent des opérations de push et de pop avec une complexité constante à chaque extrémité.

Examples

Création d'une classe Stack avec un objet List

En utilisant un objet `list` , vous pouvez créer une pile générique entièrement fonctionnelle avec des méthodes d'aide telles que la vérification et la vérification de l'empilement de la pile. Consultez la documentation officielle de Python pour utiliser la `list` comme `Stack` ici .

```
#define a stack class
class Stack:
    def __init__(self):
        self.items = []

    #method to check the stack is empty or not
    def isEmpty(self):
        return self.items == []

    #method for pushing an item
    def push(self, item):
        self.items.append(item)

    #method for popping an item
    def pop(self):
        return self.items.pop()

    #check what item is on top of the stack without removing it
    def peek(self):
        return self.items[-1]

    #method to get the size
    def size(self):
        return len(self.items)

    #to view the entire stack
    def fullStack(self):
        return self.items
```

Un exemple d'exécution:

```
stack = Stack()
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
print('Pushing integer 1')
stack.push(1)
print('Pushing string "Told you, I am generic stack!"')
stack.push('Told you, I am generic stack!')
print('Pushing integer 3')
stack.push(3)
print('Current stack:', stack.fullStack())
print('Popped item:', stack.pop())
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
```

Sortie:

```
Current stack: []
Stack empty?: True
Pushing integer 1
Pushing string "Told you, I am generic stack!"
Pushing integer 3
Current stack: [1, 'Told you, I am generic stack!', 3]
Popped item: 3
Current stack: [1, 'Told you, I am generic stack!']
Stack empty?: False
```

Parenthèses parentales

Les piles sont souvent utilisées pour l'analyse. Une tâche d'analyse simple consiste à vérifier si une chaîne de parenthèses correspond.

Par exemple, la chaîne `([])` correspond, car les crochets externes et internes forment des paires. `()<>)` ne correspond pas, car le dernier `)` n'a pas de partenaire. `([)]` ne correspond pas non plus, car les paires doivent être entièrement à l'intérieur ou à l'extérieur des autres paires.

```
def checkParenth(str):
    stack = Stack()
    pushChars, popChars = "<({[", ">)}]"
    for c in str:
        if c in pushChars:
            stack.push(c)
        elif c in popChars:
            if stack.isEmpty():
                return False
            else:
                stackTop = stack.pop()
                # Checks to see whether the opening bracket matches the closing one
                balancingBracket = pushChars[popChars.index(c)]
                if stackTop != balancingBracket:
                    return False
        else:
            return False

    return not stack.isEmpty()
```

Lire Empiler en ligne: https://riptutorial.com/fr/python/topic/3807/empiler

Chapitre 59: Enregistrement

Examples

Introduction à la journalisation Python

Ce module définit des fonctions et des classes qui implémentent un système flexible de journalisation des événements pour les applications et les bibliothèques.

Le principal avantage de disposer de l'API de journalisation fournie par un module de bibliothèque standard est que tous les modules Python peuvent participer à la journalisation. Votre journal d'application peut donc inclure vos propres messages intégrés à des messages provenant de modules tiers.

Alors, commençons:

Exemple de configuration directement dans le code

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('this is a %s test', 'debug')
```

Exemple de sortie:

```
2016-07-26 18:53:55,332 root         DEBUG    this is a debug test
```

Exemple de configuration via un fichier INI

En supposant que le fichier s'appelle logging_config.ini. Plus de détails sur le format de fichier se trouvent dans la section de configuration de la journalisation du didacticiel de journalisation .

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler
```

```
[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Ensuite, utilisez `logging.config.fileConfig()` dans le code:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Exemple de configuration via un dictionnaire

A partir de Python 2.7, vous pouvez utiliser un dictionnaire avec des détails de configuration. Le PEP 391 contient une liste des éléments obligatoires et facultatifs du dictionnaire de configuration.

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
        },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
              'formatter': 'f',
              'level': logging.DEBUG}
        },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
        },
)

dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Exceptions de journalisation

Si vous souhaitez enregistrer des exceptions, vous pouvez et devez utiliser la `logging.exception(msg)` :

```
>>> import logging
```

```
>>> logging.basicConfig()
>>> try:
...     raise Exception('foo')
... except:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

Ne passez pas l'exception en tant qu'argument:

Comme `logging.exception(msg)` attend un argument `msg` , il est courant de passer l'exception dans l'appel de journalisation comme ceci:

```
>>> try:
...     raise Exception('foo')
... except Exception as e:
...     logging.exception(e)
...
ERROR:root:foo
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

Bien que cela puisse sembler être la bonne chose à faire au début, il est en réalité problématique en raison de la façon dont les exceptions et divers codages fonctionnent ensemble dans le module de journalisation:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception(e)
...
Traceback (most recent call last):
  File "/.../python2.7/logging/__init__.py", line 861, in emit
    msg = self.format(record)
  File "/.../python2.7/logging/__init__.py", line 734, in format
    return fmt.format(record)
  File "/.../python2.7/logging/__init__.py", line 469, in format
    s = self._fmt % record.__dict__
UnicodeEncodeError: 'ascii' codec can't encode characters in position 1-2: ordinal not in
range(128)
Logged from file <stdin>, line 4
```

En essayant de connecter une exception contenant des caractères unicode, cette méthode échouera lamentablement . Il masquera le stacktrace de l'exception d'origine en le remplaçant par un nouveau qui est `logging.exception(e)` lors du formatage de votre `logging.exception(e)` .

De toute évidence, dans votre propre code, vous pourriez avoir connaissance de l'encodage des exceptions. Cependant, les bibliothèques tierces peuvent gérer cela différemment.

Usage correct:

Si au lieu de l'exception il suffit de passer un message et de laisser python faire sa magie, cela fonctionnera:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

Comme vous pouvez le voir, nous n'utilisons pas `e` dans ce cas, l'appel à `logging.exception(...)` formate comme par magie l'exception la plus récente.

Journalisation des exceptions avec des niveaux de journal non ERROR

Si vous souhaitez enregistrer une exception avec un autre niveau de journal que ERROR, vous pouvez utiliser l'argument `exc_info` des enregistreurs par défaut:

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

Accéder au message de l'exception

Sachez que les bibliothèques peuvent générer des exceptions avec des messages comme n'importe quelle chaîne de caractères unicode ou (utf-8 si vous êtes chanceux). Si vous avez vraiment besoin d'accéder au texte d'une exception, le seul moyen fiable, qui fonctionnera toujours, est d'utiliser `repr(e)` ou le formatage de la chaîne `%r` :

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('received this exception: %r' % e)
...
ERROR:root:received this exception: Exception(u'f\xf6\xf6',)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

Lire Enregistrement en ligne: https://riptutorial.com/fr/python/topic/4081/enregistrement

Chapitre 60: Ensemble

Syntaxe

- empty_set = set () # initialise un ensemble vide
- literal_set = {'foo', 'bar', 'baz'} # construit un jeu avec 3 chaînes à l'intérieur
- set_from_list = set (['foo', 'bar', 'baz']) # appelle la fonction set pour un nouvel ensemble
- set_from_iter = set (x pour x dans la plage (30)) # utilise des itérables arbitraires pour créer un ensemble
- set_from_iter = {x pour x dans [random.randint (0,10) pour i dans la plage (10)]} # notation alternative

Remarques

Les ensembles ne sont pas *ordonnés* et ont *un temps de recherche très rapide* (O (1) amorti si vous voulez obtenir des informations techniques). C'est génial à utiliser lorsque vous avez une collection de choses, l'ordre ne compte pas, et vous allez chercher des articles en nommant beaucoup. S'il est plus judicieux de rechercher des éléments par un numéro d'index, envisagez plutôt d'utiliser une liste. Si l'ordre compte, envisagez également une liste.

Les ensembles sont *modifiables* et ne peuvent donc pas être hachés. Vous ne pouvez donc pas les utiliser comme clés de dictionnaire ou les placer dans d'autres ensembles, ou n'importe où ailleurs nécessitant des types hashibles. Dans de tels cas, vous pouvez utiliser un `frozenset` immuable.

Les éléments d'un ensemble doivent être *lavables* . Cela signifie qu'ils ont une méthode `__hash__` correcte, cohérente avec `__eq__` . En général, les types mutables tels que `list` ou `set` ne sont pas gérables et ne peuvent pas être placés dans un ensemble. Si vous rencontrez ce problème, envisagez d'utiliser `dict` et les clés immuables.

Examples

Obtenez les éléments uniques d'une liste

Disons que vous avez une liste de restaurants - peut-être que vous le lisez dans un fichier. Vous vous souciez *des* restaurants *uniques* dans la liste. La meilleure façon d'obtenir des éléments uniques dans une liste est de les transformer en un ensemble:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
unique_restaurants = set(restaurants)
print(unique_restaurants)
# prints {'Chicken Chicken', "McDonald's", 'Burger King'}
```

Notez que le jeu n'est pas dans le même ordre que la liste d'origine. c'est parce que les ensembles ne sont pas *ordonnés* , tout comme les `dict` .

Cela peut facilement être transformé en une `List` avec la fonction de `list` intégrée de Python, en donnant une autre liste qui est la même liste que l'original mais sans doublons:

```
list(unique_restaurants)
# ['Chicken Chicken', "McDonald's", 'Burger King']
```

Il est également fréquent de voir cela comme une seule ligne:

```
# Removes all duplicates and returns another list
list(set(restaurants))
```

Maintenant, toutes les opérations pouvant être effectuées sur la liste d'origine peuvent être effectuées à nouveau.

Opérations sur ensembles

avec d'autres ensembles

```
# Intersection
{1, 2, 3, 4, 5}.intersection({3, 4, 5, 6})  # {3, 4, 5}
{1, 2, 3, 4, 5} & {3, 4, 5, 6}              # {3, 4, 5}

# Union
{1, 2, 3, 4, 5}.union({3, 4, 5, 6})  # {1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5} | {3, 4, 5, 6}       # {1, 2, 3, 4, 5, 6}

# Difference
{1, 2, 3, 4}.difference({2, 3, 5})  # {1, 4}
{1, 2, 3, 4} - {2, 3, 5}            # {1, 4}

# Symmetric difference with
{1, 2, 3, 4}.symmetric_difference({2, 3, 5})  # {1, 4, 5}
{1, 2, 3, 4} ^ {2, 3, 5}                      # {1, 4, 5}

# Superset check
{1, 2}.issuperset({1, 2, 3})  # False
{1, 2} >= {1, 2, 3}           # False

# Subset check
{1, 2}.issubset({1, 2, 3})  # True
{1, 2} <= {1, 2, 3}         # True

# Disjoint check
{1, 2}.isdisjoint({3, 4})  # True
{1, 2}.isdisjoint({1, 4})  # False
```

avec des éléments simples

```
# Existence check
2 in {1,2,3}      # True
4 in {1,2,3}      # False
4 not in {1,2,3}  # True

# Add and Remove
s = {1,2,3}
```

```
s.add(4)        # s == {1,2,3,4}

s.discard(3)    # s == {1,2,4}
s.discard(5)    # s == {1,2,4}

s.remove(2)     # s == {1,4}
s.remove(2)     # KeyError!
```

Les opérations d'ensemble renvoient de nouveaux ensembles, mais ont les versions en place correspondantes:

méthode	opération sur place	méthode sur place
syndicat	s | = t	mettre à jour
intersection	s & = t	intersection_update
différence	s - = t	difference_update
symétrie_différence	s ^ = t	symmetric_difference_update

Par exemple:

```
s = {1, 2}
s.update({3, 4})   # s == {1, 2, 3, 4}
```

Ensembles versus multisets

Les ensembles sont des ensembles non ordonnés d'éléments distincts. Mais parfois, nous voulons travailler avec des ensembles non ordonnés d'éléments qui ne sont pas nécessairement distincts et garder une trace des multiplicités des éléments.

Considérez cet exemple:

```
>>> setA = {'a','b','b','c'}
>>> setA
set(['a', 'c', 'b'])
```

En enregistrant les chaînes `'a'` , `'b'` , `'b'` , `'c'` dans une structure de données définie, nous avons perdu les informations sur le fait que `'b'` se produit deux fois. Enregistrer les éléments dans une liste conserverait bien sûr cette information

```
>>> listA = ['a','b','b','c']
>>> listA
['a', 'b', 'b', 'c']
```

mais une structure de données de liste introduit un ordre supplémentaire inutile qui ralentira nos calculs.

Pour implémenter des multisets, Python fournit la classe `Counter` partir du module de `collections` (à partir de la version 2.7):

Python 2.x 2.7

```
>>> from collections import Counter
>>> counterA = Counter(['a','b','b','c'])
>>> counterA
Counter({'b': 2, 'a': 1, 'c': 1})
```

`Counter` est un dictionnaire où les éléments sont stockés en tant que clés de dictionnaire et leurs comptes sont stockés en tant que valeurs de dictionnaire. Et comme tous les dictionnaires, c'est une collection non ordonnée.

Définir les opérations en utilisant des méthodes et des intégrations

Nous définissons deux ensembles `a` et `b`

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

> REMARQUE: `{1}` crée un ensemble d'un élément, mais `{}` crée un `dict` vide. La manière correcte de créer un ensemble vide est `set()` .

Intersection

`a.intersection(b)` renvoie un nouvel ensemble avec des éléments présents à la fois dans `a` et `b`

```
>>> a.intersection(b)
{3, 4}
```

syndicat

`a.union(b)` renvoie un nouvel ensemble avec des éléments présents dans `a` et `b`

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

Différence

`a.difference(b)` renvoie un nouvel ensemble avec des éléments présents dans `a` mais pas dans `b`

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
```

```
{5}
```

Différence symétrique

`a.symmetric_difference(b)` renvoie un nouvel ensemble avec des éléments présents dans `a` ou `b` mais pas dans les deux

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

NOTE : `a.symmetric_difference(b) == b.symmetric_difference(a)`

Sous-ensemble et superset

`c.issubset(a)` teste si chaque élément de `c` est dans `a` .

`a.issuperset(c)` teste si chaque élément de `c` est dans `a` .

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

Ces dernières opérations ont des opérateurs équivalents comme indiqué ci-dessous:

Méthode	Opérateur
`a.intersection(b)`	`a & b`
`a.union(b)`	`a | b`
`a.difference(b)`	`a - b`
`a.symmetric_difference(b)`	`a ^ b`
`a.issubset(b)`	`a <= b`
`a.issuperset(b)`	`a >= b`

Ensembles disjoints

Les ensembles `a` et `d` sont disjoints si aucun élément dans `a` n'est également dans `d` et vice versa.

```
>>> d = {5, 6}
>>> a.isdisjoint(b) # {2, 3, 4} are in both sets
False
>>> a.isdisjoint(d)
True

# This is an equivalent check, but less efficient
>>> len(a & d) == 0
True

# This is even less efficient
>>> a & d == set()
True
```

Test d'adhésion

La commande interne `in` les recherches de mots clés pour occurances

```
>>> 1 in a
True
>>> 6 in a
False
```

Longueur

La fonction intégrée `len()` renvoie le nombre d'éléments de l'ensemble

```
>>> len(a)
4
>>> len(b)
3
```

Ensemble de jeux

```
{{1,2}, {3,4}}
```

mène à:

```
TypeError: unhashable type: 'set'
```

Utilisez `frozenset` :

```
{frozenset({1, 2}), frozenset({3, 4})}
```

Lire Ensemble en ligne: https://riptutorial.com/fr/python/topic/497/ensemble

Chapitre 61: Entrée et sortie de base

Examples

Utiliser input () et raw_input ()

Python 2.x 2.3

`raw_input` attendra que l'utilisateur saisisse du texte, puis renvoie le résultat sous forme de chaîne.

```
foo = raw_input("Put a message here that asks the user for input")
```

Dans l'exemple ci-dessus, `foo` stockera toutes les entrées fournies par l'utilisateur.

Python 3.x 3.0

`input` attendra que l'utilisateur saisisse du texte, puis renvoie le résultat sous forme de chaîne.

```
foo = input("Put a message here that asks the user for input")
```

Dans l'exemple ci-dessus, `foo` stockera toutes les entrées fournies par l'utilisateur.

Utiliser la fonction d'impression

Python 3.x 3.0

Dans Python 3, la fonctionnalité d'impression prend la forme d'une fonction:

```
print("This string will be displayed in the output")
# This string will be displayed in the output

print("You can print \n escape characters too.")
# You can print escape characters too.
```

Python 2.x 2.3

Dans Python 2, print était à l'origine une déclaration, comme indiqué ci-dessous.

```
print "This string will be displayed in the output"
# This string will be displayed in the output

print "You can print \n escape characters too."
# You can print escape characters too.
```

Remarque: l'utilisation `from __future__ import print_function` dans Python 2 permettra aux utilisateurs d'utiliser la fonction `print()` la même manière que le code Python 3. Ceci est uniquement disponible dans Python 2.6 et supérieur.

Fonction pour demander à l'utilisateur un numéro

```
def input_number(msg, err_msg=None):
    while True:
        try:
            return float(raw_input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)


def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)
```

Et pour l'utiliser:

```
user_number = input_number("input a number: ", "that's not a number!")
```

Ou, si vous ne voulez pas de "message d'erreur":

```
user_number = input_number("input a number: ")
```

Imprimer une chaîne sans nouvelle ligne à la fin

Python 2.x 2.3

Dans Python 2.x, pour continuer une ligne avec `print` , terminez l'instruction `print` par une virgule. Il ajoutera automatiquement un espace.

```
print "Hello,",
print "World!"
# Hello, World!
```

Python 3.x 3.0

Dans Python 3.x, la fonction d' `print` a un paramètre de `end` facultatif qui est ce qu'il imprime à la fin de la chaîne donnée. Par défaut, il s'agit d'un caractère de nouvelle ligne, si équivalent à ceci:

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

Mais vous pourriez passer dans d'autres chaînes

```
print("Hello, ", end="")
```

```
print("World!")
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

Si vous voulez plus de contrôle sur la sortie, vous pouvez utiliser `sys.stdout.write` :

```
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Lire de stdin

Les programmes Python peuvent lire des pipelines Unix . Voici un exemple simple de lecture de `stdin` :

```
import sys

for line in sys.stdin:
    print(line)
```

`sys.stdin` est un flux. Cela signifie que la boucle for ne se terminera que lorsque le flux sera terminé.

Vous pouvez maintenant diriger la sortie d'un autre programme dans votre programme python comme suit:

```
$ cat myfile | python myprogram.py
```

Dans cet exemple, `cat myfile` peut être n'importe quelle commande unix qui sort en `stdout` .

Sinon, l'utilisation du module fileinput peut s'avérer utile:

```
import fileinput
for line in fileinput.input():
    process(line)
```

Entrée d'un fichier

L'entrée peut également être lue à partir de fichiers. Les fichiers peuvent être ouverts en utilisant la fonction intégrée `open` . Utiliser une syntaxe `with <command> as <name>` syntaxe `with <command> as <name>` (appelée «gestionnaire de contexte») simplifie grandement l'utilisation de `open` et l'obtention d'un handle pour le fichier:

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

Cela garantit que, lorsque l'exécution du code quitte le bloc, le fichier est automatiquement fermé.

Les fichiers peuvent être ouverts dans différents modes. Dans l'exemple ci-dessus, le fichier est ouvert en lecture seule. Pour ouvrir un fichier existant à des fins de lecture, utilisez uniquement `r` . Si vous voulez lire ce fichier en octets, utilisez `rb` . Pour ajouter des données à un fichier existant, utilisez `a` . Utilisez `w` pour créer un fichier ou écraser les fichiers existants du même nom. Vous pouvez utiliser `r+` pour ouvrir un fichier à la fois pour la lecture et l'écriture. Le premier argument de `open()` est le nom du fichier, le second est le mode. Si le mode est laissé vide, la valeur par défaut sera `r` .

```
# let's create an example file:
with open('shoppinglist.txt', 'w') as fileobj:
    fileobj.write('tomato\npasta\ngarlic')

with open('shoppinglist.txt', 'r') as fileobj:
    # this method makes a list where each line
    # of the file is an element in the list
    lines = fileobj.readlines()

print(lines)
# ['tomato\n', 'pasta\n', 'garlic']

with open('shoppinglist.txt', 'r') as fileobj:
    # here we read the whole content into one string:
    content = fileobj.read()
    # get a list of lines, just like int the previous example:
    lines = content.split('\n')

print(lines)
# ['tomato', 'pasta', 'garlic']
```

Si la taille du fichier est infime, il est prudent de lire tout le contenu du fichier en mémoire. Si le fichier est très volumineux, il est souvent préférable de lire ligne par ligne ou par morceaux et de traiter l'entrée dans la même boucle. Pour faire ça:

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

Lors de la lecture de fichiers, tenez compte des caractères de saut de ligne spécifiques au système d'exploitation. Bien que `for line in fileobj` , il est automatiquement supprimé, il est toujours prudent d'appeler `strip()` sur les lignes lues, comme indiqué ci-dessus.

Les fichiers ouverts (`fileobj` dans les exemples ci-dessus) pointent toujours vers un emplacement spécifique du fichier. Lorsqu'ils sont ouverts pour la première fois, le descripteur de fichier pointe au tout début du fichier, qui correspond à la position `0` . Le handle de fichier peut afficher sa position actuelle avec `tell` :

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

En lisant tout le contenu, la position du gestionnaire de fichiers sera pointée à la fin du fichier:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

La position du gestionnaire de fichiers peut être définie selon vos besoins:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

Vous pouvez également lire n'importe quelle longueur du contenu du fichier pendant un appel donné. Pour cela, passez un argument pour `read()` . Lorsque `read()` est appelé sans argument, il sera lu jusqu'à la fin du fichier. Si vous passez un argument, il lira ce nombre d'octets ou de caractères, en fonction du mode (`rb` et `r` respectivement):

```
# reads the next 4 characters
# starting at the current position
next4 = fileobj.read(4)
# what we got?
print(next4) # 'cucu'
# where we are now?
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 11, as we was at 7, and read 4 chars.

fileobj.close()
```

Pour démontrer la différence entre les caractères et les octets:

```
with open('shoppinglist.txt', 'r') as fileobj:
    print(type(fileobj.read())) # <class 'str'>

with open('shoppinglist.txt', 'rb') as fileobj:
    print(type(fileobj.read())) # <class 'bytes'>
```

Lire Entrée et sortie de base en ligne: https://riptutorial.com/fr/python/topic/266/entree-et-sortie-de-base

Chapitre 62: Enum

Remarques

Les énumérations ont été ajoutées à Python dans la version 3.4 par PEP 435 .

Examples

Créer un enum (Python 2.4 à 3.3)

Les énumérations ont été backportées de Python 3.4 à Python 2.4 via Python 3.3. Vous pouvez obtenir le backport enum34 de PyPI.

```
pip install enum34
```

La création d'une enum est identique à celle de Python 3.4+

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3

print(Color.red)  # Color.red
print(Color(1))  # Color.red
print(Color['red'])  # Color.red
```

Itération

Les énumérations sont itérables:

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3

[c for c in Color]  # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

Lire Enum en ligne: https://riptutorial.com/fr/python/topic/947/enum

Chapitre 63: environnement virtuel avec virtualenvwrapper

Introduction

Supposons que vous ayez besoin de travailler sur trois projets différents: le projet A, le projet B et le projet C. le projet A et le projet B nécessitent python 3 et certaines bibliothèques requises. Mais pour le projet C, vous avez besoin de python 2.7 et de bibliothèques dépendantes.

La meilleure pratique consiste donc à séparer ces environnements de projet. Pour créer un environnement virtuel, vous pouvez utiliser la technique ci-dessous:

Virtualenv, Virtualenvwrapper et Conda

Bien que nous ayons plusieurs options pour l'environnement virtuel, virtualenvwrapper est le plus recommandé.

Examples

Créer un environnement virtuel avec virtualenvwrapper

Supposons que vous ayez besoin de travailler sur trois projets différents: le projet A, le projet B et le projet C. le projet A et le projet B nécessitent python 3 et certaines bibliothèques requises. Mais pour le projet C, vous avez besoin de python 2.7 et de bibliothèques dépendantes.

La meilleure pratique consiste donc à séparer ces environnements de projet. Pour créer un environnement virtuel, vous pouvez utiliser la technique ci-dessous:

Virtualenv, Virtualenvwrapper et Conda

Bien que nous ayons plusieurs options pour l'environnement virtuel, virtualenvwrapper est le plus recommandé.

Bien que nous ayons plusieurs options pour l'environnement virtuel mais je préfère toujours virtualenvwrapper car il a plus de facilité que d'autres.

```
$ pip install virtualenvwrapper

$ export WORKON_HOME=~/Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
$ printf '\n%s\n%s\n%s' '# virtualenv' 'export WORKON_HOME=~/virtualenvs' 'source
/home/salayhin/bin/virtualenvwrapper.sh' >> ~/.bashrc
$ source ~/.bashrc

$ mkvirtualenv python_3.5
Installing
```

```
setuptools.....................................
...............................................
...............................................
.............................done.
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postactivate New
python executable in python_3.5/bin/python

(python_3.5)$ ls $WORKON_HOME
python_3.5 hook.log
```

Maintenant, nous pouvons installer des logiciels dans l'environnement.

```
(python_3.5)$ pip install django
Downloading/unpacking django
Downloading Django-1.1.1.tar.gz (5.6Mb): 5.6Mb downloaded
Running setup.py egg_info for package django
Installing collected packages: django
Running setup.py install for django
changing mode of build/scripts-2.6/django-admin.py from 644 to 755
changing mode of /Users/salayhin/Envs/env1/bin/django-admin.py to 755
Successfully installed django
```

Nous pouvons voir le nouveau paquet avec lssitepackages:

```
(python_3.5)$ lssitepackages
Django-1.1.1-py2.6.egg-info easy-install.pth
setuptools-0.6.10-py2.6.egg pip-0.6.3-py2.6.egg
django setuptools.pth
```

Nous pouvons créer plusieurs environnements virtuels si nous le souhaitons.

Basculer entre les environnements avec workon:

```
(python_3.6)$ workon python_3.5
(python_3.5)$ echo $VIRTUAL_ENV
/Users/salayhin/Envs/env1
(python_3.5)$
```

Pour quitter le virtualenv

```
$ deactivate
```

Lire environnement virtuel avec virtualenvwrapper en ligne:
https://riptutorial.com/fr/python/topic/9983/environnement-virtuel-avec-virtualenvwrapper

Chapitre 64: Environnement virtuel Python - virtualenv

Introduction

Un environnement virtuel ("virtualenv") est un outil permettant de créer des environnements Python isolés. Il conserve les dépendances requises par les différents projets dans des lieux distincts, en créant Python env virtuel pour eux. Cela résout le «projet A dépend de la version 2.xxx mais, le projet B a besoin du dilemme 2.xxx» et garde votre répertoire global de paquets de sites propre et gérable.

"virtualenv" crée un dossier contenant toutes les bibliothèques et tous les bacs nécessaires pour utiliser les packages dont un projet Python aurait besoin.

Examples

Installation

Installez virtualenv via pip / (apt-get):

```
pip install virtualenv
```

OU

```
apt-get install python-virtualenv
```

Remarque: Si vous rencontrez des problèmes de permission, utilisez sudo.

Usage

```
$ cd test_proj
```

Créer un environnement virtuel:

```
$ virtualenv test_proj
```

Pour commencer à utiliser l'environnement virtuel, il doit être activé:

```
$ source test_project/bin/activate
```

Pour quitter votre virtualenv, tapez simplement "désactiver":

```
$ deactivate
```

Installer un paquet dans votre Virtualenv

Si vous regardez le répertoire bin de votre virtualenv, vous verrez easy_install qui a été modifié pour placer les œufs et les paquets dans le répertoire site-packages de virtualenv. Pour installer une application dans votre environnement virtuel:

```
$ source test_project/bin/activate
$ pip install flask
```

Pour le moment, vous n'avez pas à utiliser sudo car les fichiers seront tous installés dans le répertoire local virtualenv site-packages. Ceci a été créé comme votre propre compte utilisateur.

Autres commandes virtualenv utiles

lsvirtualenv : Répertorie tous les environnements.

cdvirtualenv : Naviguez dans le répertoire de l'environnement virtuel actuellement activé pour parcourir ses packages de sites, par exemple.

cdsitepackages : Comme ci-dessus, mais directement dans le répertoire site-packages.

lssitepackages : affiche le contenu du répertoire site-packages.

Lire Environnement virtuel Python - virtualenv en ligne:
https://riptutorial.com/fr/python/topic/9782/environnement-virtuel-python---virtualenv

Chapitre 65: Environnements virtuels

Introduction

Un environnement virtuel est un outil permettant de conserver les dépendances requises par différents projets dans des endroits distincts, en créant des environnements Python virtuels pour eux. Il résout le «projet X dépend de la version 1.x mais, le projet Y nécessite 4.x», et garde votre répertoire global de paquets de sites propre et gérable.

Cela permet d'isoler vos environnements pour différents projets les uns des autres et de vos bibliothèques système.

Remarques

Les environnements virtuels sont suffisamment utiles pour être probablement utilisés pour chaque projet. En particulier, les environnements virtuels vous permettent de:

1. Gérer les dépendances sans nécessiter un accès root
2. Installez différentes versions de la même dépendance, par exemple lorsque vous travaillez sur différents projets avec des exigences différentes
3. Travailler avec différentes versions de python

Examples

Créer et utiliser un environnement virtuel

`virtualenv` est un outil permettant de créer des environnements Python isolés. Ce programme crée un dossier contenant tous les exécutables nécessaires pour utiliser les packages dont un projet Python aurait besoin.

Installation de l'outil virtualenv

Ceci n'est nécessaire qu'une fois. Le programme `virtualenv` peut être disponible via votre distribution. Sur les distributions de type Debian, le paquet s'appelle `python-virtualenv` ou `python3-virtualenv` .

Vous pouvez également installer `virtualenv` utilisant pip :

```
$ pip install virtualenv
```

Créer un nouvel environnement virtuel

Cela n'est nécessaire qu'une fois par projet. Lors du démarrage d'un projet pour lequel vous souhaitez isoler des dépendances, vous pouvez configurer un nouvel environnement virtuel pour ce projet:

```
$ virtualenv foo
```

Cela va créer un `foo` dossier contenant des scripts d'outillage et une copie du `python` binaire lui - même. Le nom du dossier n'est pas pertinent. Une fois l'environnement virtuel créé, il est autonome et ne nécessite aucune manipulation supplémentaire avec l'outil `virtualenv` . Vous pouvez maintenant commencer à utiliser l'environnement virtuel.

Activer un environnement virtuel existant

Pour *activer* un environnement virtuel, un shell magique est requis afin que votre Python soit celui de `foo` au lieu du système. C'est le but du fichier d' `activate` , que vous devez rechercher dans votre shell actuel:

```
$ source foo/bin/activate
```

Les utilisateurs Windows doivent taper:

```
$ foo\Scripts\activate.bat
```

Une fois qu'un environnement virtuel a été activé, les fichiers binaires `python` et `pip` et tous les scripts installés par des modules tiers sont ceux de `foo` . En particulier, tous les modules installés avec `pip` seront déployés dans l'environnement virtuel, ce qui permettra un environnement de développement contenu. L'activation de l'environnement virtuel doit également ajouter un préfixe à votre invite, comme indiqué dans les commandes suivantes.

```
# Installs 'requests' to foo only, not globally
(foo)$ pip install requests
```

Enregistrement et restauration des dépendances

Pour enregistrer les modules que vous avez installés via `pip` , vous pouvez répertorier tous ces modules (et les versions correspondantes) dans un fichier texte en utilisant la commande `freeze` . Cela permet aux autres d'installer rapidement les modules Python nécessaires à l'application en utilisant la commande install. Le nom conventionnel d'un tel fichier est `requirements.txt` :

```
(foo)$ pip freeze > requirements.txt
(foo)$ pip install -r requirements.txt
```

Veuillez noter que `freeze` répertorie tous les modules, y compris les dépendances transitives requises par les modules de niveau supérieur que vous avez installés manuellement. En tant que tel, vous pouvez préférer créer le fichier `requirements.txt` à la main , en ne mettant que les

modules de premier niveau dont vous avez besoin.

Quitter un environnement virtuel

Si vous avez fini de travailler dans l'environnement virtuel, vous pouvez le désactiver pour revenir à votre shell normal:

```
(foo)$ deactivate
```

Utilisation d'un environnement virtuel dans un hôte partagé

Parfois, il n'est pas possible de `$ source bin/activate` un virtualenv, par exemple si vous utilisez mod_wsgi dans un hôte partagé ou si vous n'avez pas accès à un système de fichiers, comme dans Amazon API Gateway ou Google AppEngine. Dans ces cas, vous pouvez déployer les bibliothèques que vous avez installées dans votre virtualenv local et patcher votre `sys.path` .

Heureusement virtualenv est livré avec un script qui met à jour votre `sys.path` et votre `sys.prefix`

```
import os

mydir = os.path.dirname(os.path.realpath(__file__))
activate_this = mydir + '/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

Vous devriez ajouter ces lignes au tout début du fichier que votre serveur exécutera.

Cela trouvera le `bin/activate_this.py` que `virtualenv` créé le fichier dans le même répertoire que vous exécutez et ajoute vos `lib/python2.7/site-packages` à `sys.path`

Si vous souhaitez utiliser le script `activate_this.py` , n'oubliez pas de déployer au moins les répertoires `bin` et `lib/python2.7/site-packages` et leur contenu.

Python 3.x 3.3

Environnements virtuels intégrés

A partir de Python 3.3, le module venv créera des environnements virtuels. La commande `pyvenv` n'a pas besoin d'être installée séparément:

```
$ pyvenv foo
$ source foo/bin/activate
```

ou

```
$ python3 -m venv foo
$ source foo/bin/activate
```

Installation de packages dans un environnement virtuel

Une fois votre environnement virtuel activé, tous les packages que vous installez seront désormais installés dans `virtualenv` et non globalement. Par conséquent, les nouveaux paquets peuvent être sans avoir besoin des privilèges root.

Pour vérifier que les packages sont installés dans `virtualenv` exécutez la commande suivante pour vérifier le chemin d'accès de l'exécutable utilisé:

```
(<Virtualenv Name) $ which python
/<Virtualenv Directory>/bin/python

(Virtualenv Name) $ which pip
/<Virtualenv Directory>/bin/pip
```

Tout paquet installé avec pip sera installé dans `virtualenv` lui-même dans le répertoire suivant:

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

Vous pouvez également créer un fichier répertoriant les packages nécessaires.

requirements.txt :

```
requests==2.10.0
```

En cours d'exécution:

```
# Install packages from requirements.txt
pip install -r requirements.txt
```

va installer la version 2.10.0 des `requests` package.

Vous pouvez également obtenir une liste des packages et de leurs versions actuellement installés dans l'environnement virtuel actif:

```
# Get a list of installed packages
pip freeze

# Output list of packages and versions into a requirement.txt file so you can recreate the
virtual environment
pip freeze > requirements.txt
```

Vous n'avez pas besoin d'activer votre environnement virtuel chaque fois que vous devez installer un package. Vous pouvez directement utiliser l'exécutable pip dans le répertoire d'environnement virtuel pour installer les packages.

```
$ /<Virtualenv Directory>/bin/pip install requests
```

Plus d'informations sur l'utilisation de pip peuvent être trouvées sur le sujet PIP .

Comme vous installez sans root dans un environnement virtuel, il *ne* s'agit *pas d'* une installation globale sur l'ensemble du système - le package installé sera uniquement disponible dans l'environnement virtuel actuel.

Créer un environnement virtuel pour une version différente de python

En supposant que `python` et `python3` soient tous deux installés, il est possible de créer un environnement virtuel pour Python 3 même si `python3` n'est pas le Python par défaut:

```
virtualenv -p python3 foo
```

ou

```
virtualenv --python=python3 foo
```

ou

```
python3 -m venv foo
```

ou

```
pyvenv foo
```

En fait, vous pouvez créer un environnement virtuel basé sur n'importe quelle version de python de votre système. Vous pouvez vérifier différents python fonctionnels sous `/usr/bin/` ou `/usr/local/bin/` (sous Linux) OU dans `/Library/Frameworks/Python.framework/Versions/XX/bin/` (OSX), puis déterminez la nommez-le et utilisez-le dans l' `--python` ou `-p` lors de la création de l'environnement virtuel.

Gestion de plusieurs environnements virtuels avec virtualenvwrapper

L'utilitaire `virtualenvwrapper` simplifie le travail avec les environnements virtuels et est particulièrement utile si vous traitez de nombreux environnements / projets virtuels.

Au lieu d'avoir à gérer les répertoires d'environnement virtuel vous-même, `virtualenvwrapper` gère pour vous, en stockant tous les environnements virtuels sous un répertoire central (`~/.virtualenvs` par défaut).

Installation

Installez `virtualenvwrapper` avec le gestionnaire de paquets de votre système.

Debian / Ubuntu:

```
apt-get install virtualenvwrapper
```

Fedora / CentOS / RHEL:

```
yum install python-virtualenvrwapper
```

Arch Linux:

```
pacman -S python-virtualenvwrapper
```

Ou installez-le depuis PyPI en utilisant `pip` :

```
pip install virtualenvwrapper
```

Sous Windows, vous pouvez utiliser `virtualenvwrapper-win` ou `virtualenvwrapper-powershell` place.

Usage

Les environnements virtuels sont créés avec `mkvirtualenv` . Tous les arguments de la commande `virtualenv` origine sont également acceptés.

```
mkvirtualenv my-project
```

ou par exemple

```
mkvirtualenv --system-site-packages my-project
```

Le nouvel environnement virtuel est automatiquement activé. Dans les nouveaux shells, vous pouvez activer l'environnement virtuel avec `workon`

```
workon my-project
```

L'avantage de la commande `workon` par rapport à la traditionnelle . `path/to/my-env/bin/activate` est que la commande `workon` fonctionnera dans n'importe quel répertoire; vous n'avez pas besoin de vous souvenir dans quel répertoire est stocké l'environnement virtuel de votre projet.

Répertoires de projets

Vous pouvez même spécifier un répertoire de projet lors de la création de l'environnement virtuel avec l'option `-a` ou une version ultérieure avec la commande `setvirtualenvproject` .

```
mkvirtualenv -a /path/to/my-project my-project
```

ou

```
workon my-project
cd /path/to/my-project
setvirtualenvproject
```

`workon` un projet, la commande `workon` bascule automatiquement sur le projet et `cdproject` commande `cdproject` qui vous permet de passer au répertoire du projet.

Pour voir une liste de tous les virtualenvs gérés par virtualenvwrapper, utilisez `lsvirtualenv` .

Pour supprimer une virtualenv, utilisez `rmvirtualenv` :

```
rmvirtualenv my-project
```

Chaque virtualenv géré par virtualenvwrapper comprend 4 scripts bash vides: `preactivate` , `postactivate` , `predeactivate` et `postdeactivate` . Celles-ci servent de crochets pour exécuter des commandes bash à certains moments du cycle de vie de la virtualenv; Par exemple, toutes les commandes du script `postactivate` s'exécuteront juste après l'activation de virtualenv. Ce serait un bon endroit pour définir des variables d'environnement spéciales, des alias ou tout autre élément pertinent. Les 4 scripts sont situés sous `.virtualenvs/<virtualenv_name>/bin/` .

Pour plus de détails, lisez la documentation de virtualenvwrapper .

Découvrir l'environnement virtuel que vous utilisez

Si vous utilisez l'invite `bash` par défaut sous Linux, vous devriez voir le nom de l'environnement virtuel au début de votre invite.

```
(my-project-env) user@hostname:~$ which python
/home/user/my-project-env/bin/python
```

Spécification de la version spécifique de python à utiliser dans un script sous Unix / Linux

Pour spécifier quelle version de python le shell Linux doit utiliser la première ligne de scripts Python peut être une ligne shebang, qui commence par `#!` :

```
#!/usr/bin/python
```

Si vous êtes dans un environnement virtuel, alors `python myscript.py` utilisera Python depuis votre environnement virtuel, mais `./myscript.py` utilisera l'interpréteur Python dans le `./myscript.py` `#!` ligne. Pour vous assurer que Python de l'environnement virtuel est utilisé, remplacez la première ligne par:

```
#!/usr/bin/env python
```

Après avoir spécifié la ligne shebang, n'oubliez pas de donner des autorisations d'exécution au script en procédant comme suit:

```
chmod +x myscript.py
```

Cela vous permettra d'exécuter le script en exécutant `./myscript.py` (ou de fournir le chemin absolu du script) au lieu de `python myscript.py` ou `python3 myscript.py` .

Utiliser virtualenv avec une coquille de poisson

Fish shell est plus convivial, mais vous pourriez rencontrer des problèmes lors de l'utilisation de `virtualenv` ou `virtualenvwrapper` . `virtualfish` existe également pour le sauvetage. Suivez simplement la séquence ci-dessous pour commencer à utiliser Fish shell avec virtualenv.

- Installer virtualfish dans l'espace global

  ```
  sudo pip install virtualfish
  ```

- Chargez le module virtuel python pendant le démarrage du shell

  ```
  $ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
  ```

- Editez cette fonction `fish_prompt` par `$ funced fish_prompt --editor vim` et ajoutez les lignes ci-dessous et fermez l'éditeur vim

  ```
  if set -q VIRTUAL_ENV
      echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color
  normal) " "
  end
  ```

  Note: Si vous n'êtes pas familier avec vim, fournissez simplement votre éditeur préféré comme ceci `$ funced fish_prompt --editor nano` ou `$ funced fish_prompt --editor gedit`

- Enregistrer les modifications à l'aide de `funcsave`

  ```
  funcsave fish_prompt
  ```

- Pour créer un nouvel environnement virtuel, utilisez `vf new`

  ```
  vf new my_new_env # Make sure $HOME/.virtualenv exists
  ```

- Si vous voulez créer un nouvel environnement python3, spécifiez-le via l' `-p`

  ```
  vf new -p python3 my_new_env
  ```

- Pour basculer entre les environnements virtuels, utilisez `vf deactivate` & `vf activate another_env`

Liens officiels:

- https://github.com/adambrenecki/virtualfish

- http://virtualfish.readthedocs.io/en/latest/

Créer des environnements virtuels avec Anaconda

Anaconda est une alternative puissante à `virtualenv` - un gestionnaire de paquets multi-plateforme, semblable à un `pip` , doté de fonctionnalités permettant de créer et de supprimer rapidement des environnements virtuels. Après avoir installé Anaconda, voici quelques commandes pour commencer:

Créer un environnement

```
conda create --name <envname> python=<version>
```

où `<envname>` dans un nom arbitraire pour votre environnement virtuel, et `<version>` est une version spécifique de Python que vous souhaitez configurer.

Activer et désactiver votre environnement

```
# Linux, Mac
source activate <envname>
source deactivate
```

ou

```
# Windows
activate <envname>
deactivate
```

Afficher une liste des environnements créés

```
conda env list
```

Supprimer un environnement

```
conda env remove -n <envname>
```

Trouvez plus de commandes et de fonctionnalités dans la documentation officielle du conda .

Vérifier s'il est exécuté dans un environnement virtuel

Parfois, l'invite du shell n'affiche pas le nom de l'environnement virtuel et vous voulez vous assurer que vous êtes dans un environnement virtuel ou non.

Exécutez l'interpréteur python et essayez:

```
import sys
sys.prefix
```

```
sys.real_prefix
```

- En dehors d'un environnement virtuel, l'environnement `sys.prefix` pointe vers l'installation python du système et `sys.real_prefix` n'est pas défini.
- Dans un environnement virtuel, `sys.prefix` pointe vers l'environnement virtuel l'installation de python et `sys.real_prefix` pointe vers l'installation python du système.

Pour les environnements virtuels créés à l'aide du module venv standard, il n'y a pas de `sys.real_prefix` . Au lieu de cela, vérifiez si `sys.base_prefix` est identique à `sys.prefix` .

Lire Environnements virtuels en ligne: https://riptutorial.com/fr/python/topic/868/environnements-virtuels

Chapitre 66: étagère

Introduction

Shelve est un module Python utilisé pour stocker des objets dans un fichier. Le module shelve implémente un stockage persistant pour des objets Python arbitraires pouvant être décapés, à l'aide d'une API de type dictionnaire. Le module shelve peut être utilisé comme une option de stockage persistant simple pour les objets Python lorsqu'une base de données relationnelle est excessive. La tablette est accessible par des touches, comme avec un dictionnaire. Les valeurs sont décapées et écrites dans une base de données créée et gérée par anydbm.

Remarques

Remarque: Ne vous fiez pas à la fermeture automatique de la tablette; appelez toujours `close()` explicitement lorsque vous n'en avez plus besoin, ou utilisez `shelve.open()` comme gestionnaire de contexte:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Attention:

Étant donné que le module de `shelve` est soutenu par un `pickle` , il est impossible de charger une étagère depuis une source non fiable. Comme avec les conserves au vinaigre, charger une étagère peut exécuter du code arbitraire.

Restrictions

1 . Le choix du package de base de données à utiliser (tel que dbm.ndbm ou dbm.gnu) dépend de l'interface disponible. Par conséquent, il n'est pas prudent d'ouvrir la base de données directement à l'aide de dbm. La base de données est également (malheureusement) soumise aux limitations de dbm, si elle est utilisée - cela signifie que (la représentation décapée de) les objets stockés dans la base de données doivent être assez petits, et dans de rares cas refuser les mises à jour

2 .Le module Shelve ne prend pas en charge les accès simultanés en lecture / écriture aux objets mis en attente. (Plusieurs accès en lecture simultanée sont sécurisés.) Lorsqu'un programme a une étagère ouverte pour l'écriture, aucun autre programme ne devrait l'ouvrir pour la lecture ou l'écriture. Le verrouillage de fichier Unix peut être utilisé pour résoudre ce problème, mais cela diffère selon les versions Unix et nécessite des connaissances sur l'implémentation de la base de données utilisée.

Examples

Exemple de code pour le rayonnage

Pour mettre un objet en mémoire, commencez par importer le module puis attribuez-lui la valeur suivante:

```
import shelve
 database = shelve.open(filename.suffix)
 object = Object()
 database['key'] = object
```

Pour résumer l'interface (key est une chaîne, data est un objet arbitraire):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

Créer une nouvelle étagère

Le moyen le plus simple d'utiliser Shelve est la classe **DbfilenameShelf** . Il utilise anydbm pour stocker les données. Vous pouvez utiliser la classe directement, ou simplement appeler **shelve.open ()** :

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
finally:
    s.close()
```

Pour accéder à nouveau aux données, ouvrez l'étagère et utilisez-la comme un dictionnaire:

```
    import shelve

    s = shelve.open('test_shelf.db')
    try:
        existing = s['key1']
    finally:
        s.close()

print existing
```

Si vous exécutez les deux exemples de scripts, vous devriez voir:

```
$ python shelve_create.py
$ python shelve_existing.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

Le module **dbm** ne prend pas en charge plusieurs applications écrivant simultanément dans la même base de données. Si vous savez que votre client ne modifiera pas l'étagère, vous pouvez demander à shelve d'ouvrir la base de données en lecture seule.

```
import shelve

s = shelve.open('test_shelf.db', flag='r')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

Si votre programme tente de modifier la base de données alors qu'il est ouvert en lecture seule, une exception d'erreur d'accès est générée. Le type d'exception dépend du module de base de données sélectionné par anydbm lors de la création de la base de données.

Réécrire

Les étagères ne suivent pas les modifications apportées aux objets volatils, par défaut. Cela signifie que si vous modifiez le contenu d'un élément stocké dans l'étagère, vous devez mettre à jour l'étagère explicitement en stockant à nouveau l'élément.

```
import shelve

s = shelve.open('test_shelf.db')
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
```

```
try:
    print s['key1']
finally:
    s.close()
```

Dans cet exemple, le dictionnaire de 'key1' n'est plus stocké, de sorte que lorsque la tablette est rouverte, les modifications n'ont pas été conservées.

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

Pour capturer automatiquement les modifications apportées aux objets volatiles stockés sur l'étagère, ouvrez l'étagère avec l'écriture différée activée. L'indicateur d'écriture différée oblige l'étagère à mémoriser tous les objets extraits de la base de données à l'aide d'un cache en mémoire. Chaque objet de cache est également réécrit dans la base de données lorsque l'étagère est fermée.

```
import shelve

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
    print s['key1']
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

Bien que cela réduise le risque d'erreur du programmeur et rend la persistance de l'objet plus transparente, l'utilisation du mode d'écriture différée peut ne pas être souhaitable dans toutes les situations. Le cache consomme de la mémoire supplémentaire lorsque l'étagère est ouverte et la pause pour écrire chaque objet mis en cache dans la base de données à sa fermeture peut prendre plus de temps. Comme il n'y a aucun moyen de savoir si les objets mis en cache ont été modifiés, ils sont tous réécrits. Si votre application lit les données plus qu'elle écrit, l'écriture différée ajoutera plus de données que vous ne le souhaiteriez.

```
$ python shelve_create.py
$ python shelve_writeback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
```

Lire étagère en ligne: https://riptutorial.com/fr/python/topic/10629/etagere

Chapitre 67: Événements envoyés par le serveur Python

Introduction

SSE (Server Sent Events) est une connexion unidirectionnelle entre un serveur et un client (généralement un navigateur Web) qui permet au serveur de "transmettre" des informations au client. C'est un peu comme les websockets et les longues interrogations. La principale différence entre SSE et websockets est que SSE est unidirectionnel, seul le serveur peut envoyer des informations au client, comme avec les websockets, les deux peuvent envoyer des informations à chacun. SSE est généralement considéré comme beaucoup plus simple à utiliser / implémenter que les Websockets.

Examples

Flacon SSE

```
@route("/stream")
def stream():
    def event_stream():
        while True:
            if message_to_send:
                yield "data:
                    {}\n\n".format(message_to_send)"

    return Response(event_stream(), mimetype="text/event-stream")
```

Asyncio SSE

Cet exemple utilise la bibliothèque asyncio SSE: https://github.com/brutasse/asyncio-sse

```
import asyncio
import sse

class Handler(sse.Handler):
    @asyncio.coroutine
    def handle_request(self):
        yield from asyncio.sleep(2)
        self.send('foo')
        yield from asyncio.sleep(2)
        self.send('bar', event='wakeup')

start_server = sse.serve(Handler, 'localhost', 8888)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Lire Événements envoyés par le serveur Python en ligne:
https://riptutorial.com/fr/python/topic/9100/evenements-envoyes-par-le-serveur-python

Chapitre 68: Exceptions du Commonwealth

Introduction

Ici, dans Stack Overflow, nous voyons souvent des doublons parlant des mêmes erreurs: `"ImportError: No module named '??????'`, `SyntaxError: invalid syntax` ou `NameError: name '???' is not defined` c'est un effort pour les réduire et avoir des documents à relier.

Examples

IndentationErrors (ou indentation SyntaxErrors)

Dans la plupart des autres langages, l'indentation n'est pas obligatoire, mais en Python (et autres langages: premières versions de FORTRAN, Makefiles, Whitespace (langage ésotérique), etc.), ce qui peut être déroutant si vous venez d'une autre langue. Si vous copiez du code d'un exemple vers le vôtre ou simplement si vous êtes nouveau.

IndentationError / SyntaxError: retrait inattendu

Cette exception est levée lorsque le niveau d'indentation augmente sans raison.

Exemple

Il n'y a aucune raison d'augmenter le niveau ici:

Python 2.x 2.0 2.7

```
print "This line is ok"
    print "This line isn't ok"
```

Python 3.x 3.0

```
print("This line is ok")
    print("This line isn't ok")
```

Ici, il y a deux erreurs: la dernière et que l'indentation ne correspond à aucun niveau d'indentation. Cependant, un seul est affiché:

Python 2.x 2.0 2.7

```
print "This line is ok"
 print "This line isn't ok"
```

Python 3.x 3.0

```
 print("This line is ok")
  print("This line isn't ok")
```

IndentationError / SyntaxError: unindent ne correspond à aucun niveau d'indentation externe

Apparaît que vous ne vous êtes pas complètement désintéressé.

Exemple

Python 2.x 2.0 2.7

```
def foo():
    print "This should be part of foo()"
   print "ERROR!"
print "This is not a part of foo()"
```

Python 3.x 3.0

```
 print("This line is ok")
  print("This line isn't ok")
```

IndentationError: attend un bloc en retrait

Après deux points (puis une nouvelle ligne), le niveau d'indentation doit augmenter. Cette erreur est soulevée lorsque cela ne s'est pas produit.

Exemple

```
if ok:
doStuff()
```

Remarque : Utilisez le mot `pass` clé `pass` (qui ne fait absolument rien) pour mettre simplement une `method` `if` , `else` , `except` , `class` , `method` ou `definition` sans indiquer ce qui se passera si appelé / condition est vrai (mais le faire plus tard ou dans le cas de `except` : il suffit de ne rien faire):

```
def foo():
    pass
```

IndentationError: utilisation incohérente des tabulations et des espaces dans l'indentation

Exemple

```
def foo():
    if ok:
      return "Two != Four != Tab"
        return "i dont care i do whatever i want"
```

Comment éviter cette erreur

N'utilisez pas d'onglets. Il est déconseillé par `PEP8` , le guide de style pour Python.

1. Configurez votre éditeur pour utiliser 4 **espaces** pour l'indentation.
2. Effectuez une recherche et remplacez pour remplacer tous les onglets par 4 espaces.
3. Assurez-vous que votre éditeur est configuré pour **afficher des** onglets de 8 espaces, afin que vous puissiez facilement réaliser cette erreur et la corriger.

Voir cette question si vous voulez en savoir plus.

TypeErrors

Ces exceptions sont provoquées lorsque le type d'un objet doit être différent

TypeError: [définition / méthode] prend? arguments positionnels mais? a été donné

Une fonction ou une méthode a été appelée avec plus (ou moins) d'arguments que ceux qu'elle peut accepter.

Exemple

Si plus d'arguments sont donnés:

```
def foo(a): return a
foo(a,b,c,d) #And a,b,c,d are defined
```

Si moins d'arguments sont donnés:

```
def foo(a,b,c,d): return a += b + c + d
```

```
foo(a) #And a is defined
```

Note : si vous voulez utiliser un nombre inconnu d'arguments, vous pouvez utiliser `*args` ou `**kwargs` . Voir * args et ** kwargs

---

TypeError: type (s) d'opérande non pris en charge pour [opérande]: '???' et '???'

Certains types ne peuvent pas être utilisés ensemble, selon l'opérande.

Exemple

Par exemple: `+` est utilisé pour concaténer et ajouter, mais vous ne pouvez en utiliser aucun pour les deux types. Par exemple, essayer de créer un `set` en concaténant (`+` ing) `'set1'` et `'tuple1'` donne l'erreur. Code:

```
set1, tuple1 = {1,2}, (3,4)
a = set1 + tuple1
```

Certains types (ex: `int` et `string`) utilisent à la fois `+` mais pour des choses différentes:

```
b = 400 + 'foo'
```

Ou ils ne peuvent même pas être utilisés pour rien:

```
c = ["a","b"] - [1,2]
```

Mais vous pouvez par exemple ajouter un `float` à un `int` :

```
d = 1 + 1.0
```

Erreur-type: '???' l'objet n'est pas itérable / inscriptible:

Pour qu'un objet puisse être itéré, il peut prendre des index séquentiels à partir de zéro jusqu'à ce que les index ne soient plus valides et qu'un `IndexError` soit généré (plus techniquement: il doit avoir une méthode `__iter__` qui retourne un `__iterator__` ou qui définit une méthode `__getitem__` ce qui a été mentionné précédemment).

Exemple

Ici, nous disons que la `bar` est le point zéro de 1. Nonsense:

```
foo = 1
bar = foo[0]
```

Ceci est une version plus discrète: Dans cet exemple `for` tente de mettre `x` à `amount[0]` , le premier élément dans un itérables mais il ne peut pas parce que le montant est un entier:

```
amount = 10
for x in amount: print(x)
```

Erreur-type: '???' l'objet n'est pas appelable

Vous définissez une variable et l'appelez plus tard (comme ce que vous faites avec une fonction ou une méthode)

Exemple

```
foo = "notAFunction"
foo()
```

NameError: name '???' n'est pas défini

Est déclenché lorsque vous avez essayé d'utiliser une variable, une méthode ou une fonction qui n'est pas initialisée (du moins pas avant). En d'autres termes, il est déclenché lorsqu'un nom local ou global demandé n'est pas trouvé. Il est possible que vous ayez mal saisi le nom de l'objet ou que vous ayez oublié d' `import` quelque chose. Aussi peut-être que c'est dans un autre domaine. Nous couvrirons ceux avec des exemples séparés.

Ce n'est tout simplement pas défini nulle part dans le code

Il est possible que vous ayez oublié de l'initialiser, surtout s'il s'agit d'une constante

```
foo   # This variable is not defined
bar() # This function is not defined
```

Peut-être que c'est défini plus tard:

```
baz()

def baz():
    pass
```

Ou il n'a pas été `import` ed:

```
#needs import math

def sqrt():
    x = float(input("Value: "))
    return math.sqrt(x)
```

Les portées Python et la règle LEGB:

La règle dite LEGB parle des portées Python. Son nom est basé sur les différentes portées, classées selon les priorités correspondantes:

```
Local → Enclosed → Global → Built-in.
```

- **L** ocal: Variables non déclarées globales ou assignées dans une fonction.
- **E** nclosing: Les variables définies dans une fonction qui est enveloppé dans une autre fonction.
- **G** lobal: Variables déclarées globales ou affectées au niveau supérieur d'un fichier.
- **B** uilt-in: Variables pré-affectées dans le module de noms intégré.

Par exemple:

```
for i in range(4):
    d = i * 2
print(d)
```

`d` est accessible car la boucle `for` ne marque pas une nouvelle étendue, mais si c'était le cas, nous aurions une erreur et son comportement serait similaire à:

```
def noaccess():
    for i in range(4):
        d = i * 2
noaccess()
print(d)
```

Python dit `NameError: name 'd' is not defined`

Autres erreurs

AssertError

La `assert` déclaration existe dans presque toutes les langues de programmation. Quand tu fais:

```
assert condition
```

ou:

```
assert condition, message
```

C'est équivalent à ceci:

```
if __debug__:
    if not condition: raise AssertionError(message)
```

Les assertions peuvent inclure un message facultatif et vous pouvez les désactiver lorsque vous avez terminé le débogage.

Remarque : la variable intégrée **debug** est True dans des circonstances normales, False lorsque l'optimisation est demandée (option de ligne de commande -O). Les tâches à **déboguer** sont illégales. La valeur de la variable intégrée est déterminée au démarrage de l'interpréteur.

KeyboardInterrupt

Erreur lorsque l'utilisateur appuie sur la touche d'interruption, normalement `Ctrl` + `C` ou `del` .

ZeroDivisionError

Vous avez essayé de calculer `1/0` qui n'est pas défini. Voir cet exemple pour trouver les diviseurs d'un nombre:

Python 2.x 2.0 2.7

```
div = float(raw_input("Divisors of: "))
for x in xrange(div+1): #includes the number itself and zero
    if div/x == div//x:
        print x, "is a divisor of", div
```

Python 3.x 3.0

```
div = int(input("Divisors of: "))
for x in range(div+1): #includes the number itself and zero
    if div/x == div//x:
        print(x, "is a divisor of", div)
```

Il déclenche `ZeroDivisionError` car la boucle `for` assigne cette valeur à `x` . Au lieu de cela, il devrait être:

Python 2.x 2.0 2.7

```
div = float(raw_input("Divisors of: "))
for x in xrange(1,div+1): #includes the number itself but not zero
    if div/x == div//x:
        print x, "is a divisor of", div
```

Python 3.x 3.0

```
div = int(input("Divisors of: "))
for x in range(1,div+1): #includes the number itself but not zero
    if div/x == div//x:
        print(x, "is a divisor of", div)
```

Erreur de syntaxe sur un bon code

La plupart du temps, une erreur de syntaxe qui pointe vers une ligne inintéressante signifie qu'il y a un problème sur la ligne avant (dans cet exemple, il manque une parenthèse):

```
def my_print():
    x = (1 + 1
    print(x)
```

Résultats

```
  File "<input>", line 3
    print(x)
        ^
SyntaxError: invalid syntax
```

La raison la plus courante de ce problème est que les parenthèses / crochets ne correspondent pas, comme le montre l'exemple.

Il y a une mise en garde majeure pour les instructions d'impression dans Python 3:

Python 3.x 3.0

```
 >>> print "hello world"
  File "<stdin>", line 1
    print "hello world"
                      ^
SyntaxError: invalid syntax
```

Parce que l'instruction `print` été remplacée par la fonction `print()` , vous souhaitez donc:

```
print("hello world")  # Note this is valid for both Py2 & Py3
```

Lire Exceptions du Commonwealth en ligne: https://riptutorial.com/fr/python/topic/9300/exceptions-du-commonwealth

Chapitre 69: Exécution de code dynamique avec `exec` et` eval`

Syntaxe

- eval (expression [, globals = None [, locaux = None]])
- exec (objet)
- exec (objet, globales)
- exec (objet, globales, locaux)

Paramètres

Argument	Détails
`expression`	Le code de l'expression sous la forme d'une chaîne ou d'un objet de `code`
`object`	Le code d'instruction sous la forme d'une chaîne ou d'un objet de `code`
`globals`	Le dictionnaire à utiliser pour les variables globales. Si les locaux ne sont pas spécifiés, cela est également utilisé pour les locaux. Si omis, les `globals()` de la portée appelante sont utilisés.
`locals`	Un objet de *mappage* utilisé pour les variables locales. Si omis, celui passé pour les `globals` est utilisé à la place. Si les deux sont omis, les `globals()` et `locals()` de la portée d'appel sont utilisés respectivement pour les `globals` et les `locals` .

Remarques

Dans `exec` , si les `globals` sont des `locals` (c'est-à-dire qu'ils font référence au même objet), le code est exécuté comme s'il était au niveau du module. Si les `globals` et les `locals` sont des objets distincts, le code est exécuté comme s'il était dans un *corps de classe* .

Si l'objet `globals` est transmis mais ne spécifie pas la clé `__builtins__` , les fonctions et les noms intégrés à Python sont automatiquement ajoutés à la portée globale. Pour supprimer la disponibilité des fonctions telles que l' `print` ou `isinstance` dans le cadre exécuté, laissez - `globals` ont la clé `__builtins__` mis en correspondance avec la valeur `None` . Cependant, ce n'est pas une fonctionnalité de sécurité.

La syntaxe spécifique à Python 2 ne doit pas être utilisée; la syntaxe Python 3 fonctionnera dans Python 2. Ainsi, les formes suivantes sont obsolètes: <s>

- `exec object`
- `exec object in globals`
- `exec object in globals, locals`

Examples

Évaluation des instructions avec exec

```
>>> code = """for i in range(5):\n    print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Evaluer une expression avec eval

```
>>> expression = '5 + 3 * a'
>>> a = 5
>>> result = eval(expression)
>>> result
20
```

Précompiler une expression pour l'évaluer plusieurs fois

`compile` fonction intégrée de `compile` peut être utilisée pour précompiler une expression dans un objet de code; Cet objet de code peut ensuite être passé à eval. Cela accélérera les exécutions répétées du code évalué. Le 3ème paramètre à `compile` doit être la chaîne `'eval'` .

```
>>> code = compile('a * b + c', '<string>', 'eval')
>>> code
<code object <module> at 0x7f0e51a58830, file "<string>", line 1>
>>> a, b, c = 1, 2, 3
>>> eval(code)
5
```

Évaluation d'une expression avec eval à l'aide de globales personnalisés

```
>>> variables = {'a': 6, 'b': 7}
>>> eval('a * b', globals=variables)
42
```

En plus, le code ne peut pas se référer accidentellement aux noms définis à l'extérieur:

```
>>> eval('variables')
{'a': 6, 'b': 7}
>>> eval('variables', globals=variables)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'variables' is not defined
```

Utiliser `defaultdict` permet par exemple d'avoir des variables indéfinies définies à zéro:

```
>>> from collections import defaultdict
>>> variables = defaultdict(int, {'a': 42})
>>> eval('a * c', globals=variables)  # note that 'c' is not explicitly defined
0
```

Evaluer une chaîne contenant un littéral Python avec ast.literal_eval

Si vous avez une chaîne qui contient des littéraux Python, tels que des chaînes, des flottants, etc., vous pouvez utiliser `ast.literal_eval` pour évaluer sa valeur au lieu de `eval` . Cela a pour caractéristique supplémentaire de n'autoriser qu'une certaine syntaxe.

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

Cependant, ceci n'est pas sécurisé pour l'exécution du code fourni par un utilisateur non fiable, et il est trivial d'interrompre un interpréteur avec une entrée soigneusement conçue.

```
>>> import ast
>>> ast.literal_eval('()' * 1000000)
[5]    21358 segmentation fault (core dumped)  python3
```

Ici, l'entrée est une chaîne de caractères `()` répétée un million de fois, ce qui provoque un blocage de l'analyseur CPython. Les développeurs CPython ne considèrent pas les bogues de l'analyseur comme des problèmes de sécurité.

Code d'exécution fourni par un utilisateur non approuvé à l'aide de exec, eval ou ast.literal_eval

Il n'est pas possible d'utiliser `eval` ou `exec` pour exécuter du code à partir d'utilisateurs non fiables en toute sécurité. Même `ast.literal_eval` est susceptible de se `ast.literal_eval` dans l'analyseur. Il est parfois possible de se prémunir contre l'exécution de code malveillant, mais cela n'exclut pas la possibilité d'un blocage brutal de l'analyseur ou du tokenizer.

Pour évaluer le code par un utilisateur non fiable, vous devez vous tourner vers un module tiers, ou peut-être écrire votre propre analyseur syntaxique et votre propre machine virtuelle en Python.

Lire Exécution de code dynamique avec `exec` et` eval` en ligne:
https://riptutorial.com/fr/python/topic/2251/execution-de-code-dynamique-avec--exec--et--eval-

Chapitre 70: Exponentiation

Syntaxe

- valeur1 ** valeur2
- pow (valeur1, valeur2 [, valeur3])
- valeur1 .__ pow __ (valeur2 [, valeur3])
- valeur2 .__ rpow __ (valeur1)
- operator.pow (value1, value2)
- opérateur .__ pow __ (valeur1, valeur2)
- math.pow (valeur1, valeur2)
- math.sqrt (value1)
- math.exp (valeur1)
- cmath.exp (value1)
- math.expm1 (valeur1)

Examples

Racine carrée: math.sqrt () et cmath.sqrt

Le module `math` contient la fonction `math.sqrt()` qui peut calculer la racine carrée de n'importe quel nombre (qui peut être converti en un `float`) et le résultat sera toujours un `float` :

```
import math

math.sqrt(9)                 # 3.0
math.sqrt(11.11)             # 3.3331666624997918
math.sqrt(Decimal('6.25'))   # 2.5
```

La fonction `math.sqrt()` déclenche une `ValueError` si le résultat est `complex` :

```
math.sqrt(-10)
```

ValueError: erreur de domaine mathématique

`math.sqrt(x)` est *plus rapide* que `math.pow(x, 0.5)` ou `x ** 0.5` mais la précision des résultats est la même. Le module `cmath` est extrêmement similaire au module `math` , sauf qu'il peut calculer des nombres complexes et que tous ses résultats sont sous la forme d'un + bi. Il peut aussi utiliser `.sqrt()` :

```
import cmath

cmath.sqrt(4)  # 2+0j
cmath.sqrt(-4) # 2j
```

C'est quoi le `j` ? `j` est l'équivalent de la racine carrée de -1. Tous les nombres peuvent être mis

sous la forme a + bi, ou dans ce cas, a + bj. `a` est la partie réelle du nombre comme le 2 en `2+0j` . Comme il n'a pas de partie imaginaire, `b` vaut 0. `b` représente une partie de la partie imaginaire du nombre comme le 2 en `2j` . Comme il n'y a pas de véritable partie, `2j` peut aussi être écrit `0 + 2j` .

Exponentiation à l'aide des commandes intégrées: ** et pow ()

L'exponentiation peut être utilisée en utilisant la fonction `pow` intégrée ou l'opérateur `**` :

```
2 ** 3    # 8
pow(2, 3) # 8
```

Pour la plupart des opérations arithmétiques (toutes en Python 2.x), le type de résultat sera celui de l'opérande plus large. Ce n'est pas vrai pour `**` ; les cas suivants sont des exceptions à cette règle:

- Base: `int` , exposant: `int < 0` :

  ```
  2 ** -3
  # Out: 0.125 (result is a float)
  ```

- Ceci est également valable pour Python 3.x.
- Avant Python 2.2.0, cela `ValueError` une `ValueError` .
- Base: `int < 0` ou `float < 0` , exposant: `float != int`

  ```
  (-2) ** (0.5)  # also (-2.) ** (0.5)
  # Out: (8.659560562354934e-17+1.4142135623730951j) (result is complex)
  ```

- Avant python 3.0.0, cela `ValueError` une `ValueError` .

Le module d' `operator` contient deux fonctions équivalentes à l'opérateur `**` :

```
import operator
operator.pow(4, 2)      # 16
operator.__pow__(4, 3)  # 64
```

ou on pourrait appeler directement la méthode `__pow__` :

```
val1, val2 = 4, 2
val1.__pow__(val2)      # 16
val2.__rpow__(val1)     # 16
# in-place power operation isn't supported by immutable classes like int, float, complex:
# val1.__ipow__(val2)
```

Exponentiation utilisant le module mathématique: math.pow ()

Le module `math` contient une autre fonction `math.pow()` . La différence avec la fonction intégrée `pow()` ou `**` est que le résultat est toujours un `float` :

```
import math
math.pow(2, 2)    # 4.0
math.pow(-2., 2)  # 4.0
```

Ce qui exclut les calculs avec des entrées complexes:

```
math.pow(2, 2+0j)
```

    TypeError: impossible de convertir un complexe en float

et des calculs qui conduiraient à des résultats complexes:

```
math.pow(-2, 0.5)
```

    ValueError: erreur de domaine mathématique

Fonction exponentielle: math.exp () et cmath.exp ()

Les deux `math` et `cmath` -module contiennent le numéro d'Euler: e et son utilisation avec la fonction intégrée `pow()` ou `**` -operator fonctionne principalement comme `math.exp()` :

```
import math

math.e ** 2  # 7.3890560989306495
math.exp(2)  # 7.38905609893065

import cmath
cmath.e ** 2 # 7.3890560989306495
cmath.exp(2) # (7.38905609893065+0j)
```

Cependant, le résultat est différent et l'utilisation directe de la fonction exponentielle est plus fiable que l'exponentiation intégrée avec base `math.e` :

```
print(math.e ** 10)       # 22026.465794806703
print(math.exp(10))       # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
#     difference starts here ---------------^
```

Fonction exponentielle moins 1: math.expm1 ()

Le module `math` contient la fonction `expm1()` qui peut calculer l'expression `math.e ** x - 1` pour de très petits `x` avec une précision supérieure à `math.exp(x)` de `math.exp(x)` ou `cmath.exp(x)` :

```
import math

print(math.e ** 1e-3 - 1)  # 0.0010005001667083846
print(math.exp(1e-3) - 1)  # 0.0010005001667083846
print(math.expm1(1e-3))    # 0.0010005001667083417
#                            ------------------^
```

Pour très petit x la différence est plus grande:

```
print(math.e ** 1e-15 - 1) # 1.1102230246251565e-15
print(math.exp(1e-15) - 1) # 1.1102230246251565e-15
print(math.expm1(1e-15))   # 1.0000000000000007e-15
#                            ^-------------------
```

L'amélioration est significative en informatique scientifique. Par exemple, la loi de Planck contient une fonction exponentielle moins 1:

```
def planks_law(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * math.expm1(h * c / (lambda_ * k * T)))

def planks_law_naive(lambda_, T):
    from scipy.constants import h, k, c  # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * (math.e ** (h * c / (lambda_ * k * T)) - 1))

planks_law(100, 5000)        # 4.139080074896474e-19
planks_law_naive(100, 5000)  # 4.139080073488451e-19
#                                      ^----------

planks_law(1000, 5000)       # 4.139080128493406e-23
planks_law_naive(1000, 5000) # 4.139080233183142e-23
#                                    ^------------
```

Méthodes magiques et exponentiation: intégré, math et cmath

En supposant que vous ayez une classe qui stocke des valeurs uniquement entières:

```
class Integer(object):
    def __init__(self, value):
        self.value = int(value) # Cast to an integer

    def __repr__(self):
        return '{cls}({val})'.format(cls=self.__class__.__name__,
                                     val=self.value)

    def __pow__(self, other, modulo=None):
        if modulo is None:
            print('Using __pow__')
            return self.__class__(self.value ** other)
        else:
            print('Using __pow__ with modulo')
            return self.__class__(pow(self.value, other, modulo))

    def __float__(self):
        print('Using __float__')
        return float(self.value)

    def __complex__(self):
        print('Using __complex__')
        return complex(self.value, 0)
```

L'utilisation de la fonction `pow` intégrée ou de l'opérateur `**` appelle toujours `__pow__` :

```
Integer(2) ** 2                # Integer(4)
# Prints: Using __pow__
```

```
Integer(2) ** 2.5               # Integer(5)
# Prints: Using __pow__
pow(Integer(2), 0.5)            # Integer(1)
# Prints: Using __pow__
operator.pow(Integer(2), 3)     # Integer(8)
# Prints: Using __pow__
operator.__pow__(Integer(3), 3) # Integer(27)
# Prints: Using __pow__
```

Le second argument de la `__pow__()` ne peut être fourni que par l'utilisation de la méthode `pow()` ou en appelant directement la méthode:

```
pow(Integer(2), 3, 4)           # Integer(0)
# Prints: Using __pow__ with modulo
Integer(2).__pow__(3, 4)        # Integer(0)
# Prints: Using __pow__ with modulo
```

Alors que les fonctions `math` convertissent toujours en `float` et utilisent le calcul flottant:

```
import math

math.pow(Integer(2), 0.5) # 1.4142135623730951
# Prints: Using __float__
```

`cmath` cmath tentent de le convertir en `complex` mais peuvent également basculer en mode `float` s'il n'y a pas de conversion explicite en `complex` :

```
import cmath

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __complex__

del Integer.__complex__   # Deleting __complex__ method - instances cannot be cast to complex

cmath.exp(Integer(2))     # (7.38905609893065+0j)
# Prints: Using __float__
```

Ni `math` ni `cmath` ne fonctionneront si la `__float__()` est également manquante:

```
del Integer.__float__  # Deleting __complex__ method

math.sqrt(Integer(2))  # also cmath.exp(Integer(2))
```

TypeError: un float est requis

Exponentiation modulaire: pow () avec 3 arguments

Fournir `pow()` avec 3 arguments `pow(a, b, c)` évalue l' exponentiation modulaire $a^b \bmod c$:

```
pow(3, 4, 17)   # 13

# equivalent unoptimized expression:
3 ** 4 % 17     # 13
```

```
# steps:
3 ** 4          # 81
81 % 17         # 13
```

Pour les types intégrés utilisant une exponentiation modulaire n'est possible que si:

- Le premier argument est un `int`
- Le second argument est un `int >= 0`
- Le troisième argument est un `int != 0`

Ces restrictions sont également présentes dans python 3.x

Par exemple, on peut utiliser la forme à trois arguments de `pow` pour définir une fonction inverse modulaire :

```
def modular_inverse(x, p):
    """Find a such as  a·x ≡ 1 (mod p), assuming p is prime."""
    return pow(x, p-2, p)

[modular_inverse(x, 13) for x in range(1,13)]
# Out: [1, 7, 9, 10, 8, 11, 2, 5, 3, 4, 6, 12]
```

Racines: racine nième avec exposants fractionnaires

Bien que la fonction `math.sqrt` soit fournie pour le cas spécifique des racines carrées, il est souvent pratique d'utiliser l'opérateur d'exponentiation (`**`) avec des exposants fractionnaires pour effectuer des opérations nth-root, comme les racines de cube.

L'inverse d'une exponentiation est une exponentiation par la réciproque de l'exposant. Donc, si vous pouvez cuber un nombre en le plaçant à l'exposant de 3, vous pouvez trouver la racine du cube d'un nombre en le mettant à l'exposant de 1/3.

```
>>> x = 3
>>> y = x ** 3
>>> y
27
>>> z = y ** (1.0 / 3)
>>> z
3.0
>>> z == x
True
```

Calculer de grandes racines entières

Même si Python prend en charge nativement les grands nombres entiers, prendre la nième racine de très grands nombres peut échouer en Python.

```
x = 2 ** 100
cube = x ** 3
root = cube ** (1.0 / 3)
```

OverflowError: long int trop grand pour être converti en float

Lorsque vous traitez de tels entiers, vous devrez utiliser une fonction personnalisée pour calculer la nième racine d'un nombre.

```
def nth_root(x, n):
    # Start with some reasonable bounds around the nth root.
    upper_bound = 1
    while upper_bound ** n <= x:
        upper_bound *= 2
    lower_bound = upper_bound // 2
    # Keep searching for a better result as long as the bounds make sense.
    while lower_bound < upper_bound:
        mid = (lower_bound + upper_bound) // 2
        mid_nth = mid ** n
        if lower_bound < mid and mid_nth < x:
            lower_bound = mid
        elif upper_bound > mid and mid_nth > x:
            upper_bound = mid
        else:
            # Found perfect nth root.
            return mid
    return mid + 1

x = 2 ** 100
cube = x ** 3
root = nth_root(cube, 3)
x == root
# True
```

Lire Exponentiation en ligne: https://riptutorial.com/fr/python/topic/347/exponentiation

Chapitre 71: Expressions idiomatiques

Examples

Initialisations de clé de dictionnaire

Préférez la méthode `dict.get` si vous n'êtes pas sûr si la clé est présente. Il vous permet de retourner une valeur par défaut si la clé est introuvable. La méthode traditionnelle `dict[key]` `KeyError` une exception `KeyError` .

Plutôt que de faire

```
def add_student():
    try:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

Faire

```
def add_student():
        students['count'] = students.get('count', 0) + 1
```

Changement de variables

Pour changer la valeur de deux variables, vous pouvez utiliser la décompression de tuple.

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

Utilisez des tests de valeur de vérité

Python convertira implicitement n'importe quel objet en valeur booléenne pour le tester, donc utilisez-le autant que possible.

```
# Good examples, using implicit truth testing
if attr:
    # do something

if not attr:
    # do something

# Bad examples, using specific types
if attr == 1:
```

```
    # do something

if attr == True:
    # do something

if attr != '':
    # do something

# If you are looking to specifically check for None, use 'is' or 'is not'
if attr is None:
    # do something
```

Cela produit généralement un code plus lisible et est généralement beaucoup plus sûr lorsqu'il s'agit de types inattendus.

Cliquez ici pour une liste de ce qui sera évalué à `False` .

Test de "__main__" pour éviter l'exécution de code inattendue

Il est `__name__` de `__name__` variable `__name__` du programme appelant avant d'exécuter votre code.

```
import sys

def main():
    # Your code starts here

    # Don't forget to provide a return code
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

L'utilisation de ce modèle garantit que votre code n'est exécuté que lorsque vous vous attendez à ce qu'il soit; Par exemple, lorsque vous exécutez explicitement votre fichier:

```
python my_program.py
```

L'avantage, cependant, vient si vous décidez d' `import` votre fichier dans un autre programme (par exemple, si vous l'écrivez dans le cadre d'une bibliothèque). Vous pouvez ensuite `import` votre fichier et le piège `__main__` garantira qu'aucun code n'est exécuté de manière inattendue:

```
# A new program file
import my_program        # main() is not run

# But you can run main() explicitly if you really want it to run:
my_program.main()
```

Lire Expressions idiomatiques en ligne: https://riptutorial.com/fr/python/topic/3070/expressions-idiomatiques

Chapitre 72: Expressions régulières (Regex)

Introduction

Python rend les expressions régulières disponibles via le module `re` .

Les expressions régulières sont des combinaisons de caractères interprétées comme des règles pour faire correspondre les sous-chaînes. Par exemple, l'expression `'amount\D+\d+'` correspondra à toute chaîne composée du `amount` du mot plus un nombre entier, séparés par un ou plusieurs non-chiffres, tels que: `amount=100` , `amount is 3` , `amount is equal to: 33` , etc.

Syntaxe

- **Expressions régulières directes**
- re.match (pattern, string, flag = 0) # Out: correspond à un motif au début de la chaîne ou à None
- re.search (pattern, string, flag = 0) # Out: correspond à un motif à l'intérieur d'une chaîne ou à aucun
- re.findall (pattern, string, flag = 0) # Out: liste de toutes les correspondances de pattern dans string ou []
- re.finditer (pattern, string, flag = 0) # Out: identique à re.findall, mais retourne un objet d'itérateur
- re.sub (pattern, replacement, string, flag = 0) # Out: chaîne avec remplacement (chaîne ou fonction) à la place du modèle
- **Expressions régulières précompilées**
- precompiled_pattern = re.compile (pattern, flag = 0)
- precompiled_pattern.match (string) # Out: correspond au début de la chaîne ou None
- precompiled_pattern.search (string) # Out: correspond à la chaîne ou à None
- precompiled_pattern.findall (string) # Out: liste de toutes les sous-chaînes correspondantes
- precompiled_pattern.sub (chaîne / modèle / fonction, chaîne) # Out: chaîne remplacée

Examples

Faire correspondre le début d'une chaîne

Le premier argument de `re.match()` est l'expression régulière, la seconde est la chaîne à

rechercher:

```
import re

pattern = r"123"
string = "123zzb"

re.match(pattern, string)
# Out: <_sre.SRE_Match object; span=(0, 3), match='123'>

match = re.match(pattern, string)

match.group()
# Out: '123'
```

Vous remarquerez peut-être que la variable de modèle est une chaîne avec le préfixe `r` , qui indique que la chaîne est un *littéral de chaîne brut* .

Une chaîne brute littérale a une syntaxe légèrement différente de celle d' une chaîne littérale, à savoir une barre oblique inverse `\` dans un moyen littéral de chaîne brute « juste une barre oblique inverse » et il n'y a pas besoin de doubler contrecoups pour échapper à « échapper à des séquences » telles que les nouvelles lignes (`\n`) , onglets (`\t`), backspaces (`\`), flux de formulaire (`\r`), etc. Dans les littéraux de chaîne normaux, chaque barre oblique inverse doit être doublée pour éviter d'être considérée comme le début d'une séquence d'échappement.

Par conséquent, `r"\n"` est une chaîne de 2 caractères: `\` et `n` . Les modèles d'expressions rationnelles utilisent également des barres obliques inverses, par exemple `\d` fait référence à n'importe quel caractère numérique. Nous pouvons éviter de devoir échapper à nos chaînes (`"\\d"`) en utilisant des chaînes brutes (`r"\d"`).

Par exemple:

```
string = "\\t123zzb" # here the backslash is escaped, so there's no tab, just '\' and 't'
pattern = "\\t123"   # this will match \t (escaping the backslash) followed by 123
re.match(pattern, string).group()   # no match
re.match(pattern, "\t123zzb").group()  # matches '\t123'

pattern = r"\\t123"
re.match(pattern, string).group()   # matches '\\t123'
```

La correspondance est faite depuis le début de la chaîne uniquement. Si vous souhaitez faire correspondre n'importe où, utilisez plutôt `re.search` :

```
match = re.match(r"(123)", "a123zzb")

match is None
# Out: True

match = re.search(r"(123)", "a123zzb")

match.group()
# Out: '123'
```

Recherche

```
pattern = r"(your base)"
sentence = "All your base are belong to us."

match = re.search(pattern, sentence)
match.group(1)
# Out: 'your base'

match = re.search(r"(belong.*)", sentence)
match.group(1)
# Out: 'belong to us.'
```

La recherche se fait n'importe où dans la chaîne, contrairement à `re.match` . Vous pouvez également utiliser `re.findall` .

Vous pouvez également rechercher au début de la chaîne (utilisez `^`),

```
match = re.search(r"^123", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^123", "a123zzb")
match is None
# Out: True
```

à la fin de la chaîne (utilisez `$`),

```
match = re.search(r"123$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"123$", "123zzb")
match is None
# Out: True
```

ou les deux (utilisez les deux `^` et `$`):

```
match = re.search(r"^123$", "123")
match.group(0)
# Out: '123'
```

Regroupement

Le regroupement se fait entre parenthèses. Le `group()` appelant `group()` renvoie une chaîne formée des sous-groupes entre parenthèses correspondants.

```
match.group() # Group without argument returns the entire match found
# Out: '123'
match.group(0) # Specifying 0 gives the same result as specifying no argument
# Out: '123'
```

Des arguments peuvent également être fournis à `group()` pour récupérer un sous-groupe particulier.

De la documentation :

> S'il y a un seul argument, le résultat est une chaîne unique; s'il y a plusieurs arguments, le résultat est un tuple avec un élément par argument.

L'appel des `groups()` en revanche, renvoie une liste de tuples contenant les sous-groupes.

```
sentence = "This is a phone number 672-123-456-9910"
pattern = r".*(phone).*?([\d-]+)"

match = re.match(pattern, sentence)

match.groups()   # The entire match as a list of tuples of the paranthesized subgroups
# Out: ('phone', '672-123-456-9910')

m.group()        # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(0)       # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(1)       # The first parenthesized subgroup.
# Out: 'phone'

m.group(2)       # The second parenthesized subgroup.
# Out: '672-123-456-9910'

m.group(1, 2)    # Multiple arguments give us a tuple.
# Out: ('phone', '672-123-456-9910')
```

Groupes nommés

```
match = re.search(r'My name is (?P<name>[A-Za-z ]+)', 'My name is John Smith')
match.group('name')
# Out: 'John Smith'

match.group(1)
# Out: 'John Smith'
```

Crée un groupe de capture pouvant être référencé par nom et par index.

Groupes non capturés

L'utilisation de `(?:)` crée un groupe, mais le groupe n'est pas capturé. Cela signifie que vous pouvez l'utiliser en tant que groupe, mais cela ne polluera pas votre "espace de groupe".

```
re.match(r'(\d+)(\+(\d+))?', '11+22').groups()
# Out: ('11', '+22', '22')
```

```
re.match(r'(\d+)(?:\+(\d+))?', '11+22').groups()
# Out: ('11', '22')
```

Cet exemple correspond à `11+22` ou `11` , mais pas à `11+` . C'est depuis que le signe `+` et le second terme sont regroupés. Par contre, le signe `+` n'est pas capturé.

Échapper aux caractères spéciaux

Les caractères spéciaux (comme les accolades `[` et `]` ci-dessous) ne correspondent pas littéralement:

```
match = re.search(r'[b]', 'a[b]c')
match.group()
# Out: 'b'
```

En échappant aux caractères spéciaux, ils peuvent littéralement correspondre:

```
match = re.search(r'\[b\]', 'a[b]c')
match.group()
# Out: '[b]'
```

La fonction `re.escape()` peut être utilisée pour cela:

```
re.escape('a[b]c')
# Out: 'a\\[b\\]c'
match = re.search(re.escape('a[b]c'), 'a[b]c')
match.group()
# Out: 'a[b]c'
```

La fonction `re.escape()` échappe à tous les caractères spéciaux, il est donc utile que vous composiez une expression régulière en fonction de la saisie de l'utilisateur:

```
username = 'A.C.'  # suppose this came from the user
re.findall(r'Hi {}!'.format(username), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!', 'Hi ABCD!']
re.findall(r'Hi {}!'.format(re.escape(username)), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.!']
```

Remplacer

Des remplacements peuvent être effectués sur des chaînes utilisant `re.sub` .

Remplacement des chaînes

```
re.sub(r"t[0-9][0-9]", "foo", "my name t13 is t44 what t99 ever t44")
# Out: 'my name foo is foo what foo ever foo'
```

Utiliser des références de groupe

Les remplacements avec un petit nombre de groupes peuvent être faits comme suit:

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Cependant, si vous créez un identifiant de groupe tel que "10", cela ne fonctionne pas : `\10` est lu comme "ID numéro 1 suivi de 0". Il faut donc être plus précis et utiliser la notation `\g<i>` :

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

Utiliser une fonction de remplacement

```
items = ["zero", "one", "two"]
re.sub(r"a\[([0-3])\]", lambda match: items[int(match.group(1))], "Items: a[0], a[1],
something, a[2]")
# Out: 'Items: zero, one, something, two'
```

Trouver tous les matchs qui ne se chevauchent pas

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Out: ['12', '945', '444', '558', '889']
```

Notez que le `r` avant `"[0-9]{2,3}"` indique à python d'interpréter la chaîne telle quelle; comme une chaîne "brute".

Vous pouvez également utiliser `re.finditer()` qui fonctionne de la même manière que `re.findall()` mais renvoie un itérateur avec des objets `SRE_Match` au lieu d'une liste de chaînes:

```
results = re.finditer(r"([0-9]{2,3})", "some 1 text 12 is 945 here 4445588899")
print(results)
# Out: <callable-iterator object at 0x105245890>
for result in results:
     print(result.group(0))
''' Out:
12
945
444
558
889
'''
```

Motifs précompilés

```
import re
```

```
precompiled_pattern = re.compile(r"(\d+)")
matches = precompiled_pattern.search("The answer is 41!")
matches.group(1)
# Out: 41

matches = precompiled_pattern.search("Or was it 42?")
matches.group(1)
# Out: 42
```

Compiler un modèle permet de le réutiliser ultérieurement dans un programme. Cependant, notez que Python met en cache les expressions récemment utilisées (docs , réponse SO), donc *"les programmes qui n'utilisent que quelques expressions régulières à la fois n'ont pas à se soucier de la compilation des expressions régulières"* .

```
import re

precompiled_pattern = re.compile(r"(.*\d+)")
matches = precompiled_pattern.match("The answer is 41!")
print(matches.group(1))
# Out: The answer is 41

matches = precompiled_pattern.match("Or was it 42?")
print(matches.group(1))
# Out: Or was it 42
```

Il peut être utilisé avec re.match ().

Vérification des caractères autorisés

Si vous voulez vérifier qu'une chaîne ne contient qu'un certain ensemble de caractères, dans ce cas az, AZ et 0-9, vous pouvez le faire comme ceci,

```
import re

def is_allowed(string):
    characherRegex = re.compile(r'[^a-zA-Z0-9.]')
    string = characherRegex.search(string)
    return not bool(string)

print (is_allowed("abyzABYZ0099"))
# Out: 'True'

print (is_allowed("#*@#$%^"))
# Out: 'False'
```

Vous pouvez également adapter la ligne d'expression de `[^a-zA-Z0-9.] [^a-z0-9.]` , Pour interdire par exemple les lettres majuscules.

Crédit partiel: http://stackoverflow.com/a/1325265/2697955

Fractionnement d'une chaîne à l'aide d'expressions régulières

Vous pouvez également utiliser des expressions régulières pour diviser une chaîne. Par exemple,

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print( data )
# Output: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

Les drapeaux

Pour certains cas particuliers, nous devons modifier le comportement de l'expression régulière, en utilisant les indicateurs. Les `flags` peuvent être définis de deux manières, via le mot-clé `flags` ou directement dans l'expression.

Mot-clé Drapeaux

Ci-dessous un exemple pour `re.search` mais cela fonctionne pour la plupart des fonctions du module `re` .

```
m = re.search("b", "ABC")
m is None
# Out: True

m = re.search("b", "ABC", flags=re.IGNORECASE)
m.group()
# Out: 'B'

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE)
m is None
# Out: True

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE|re.DOTALL)
m.group()
# Out: 'A\nB'
```

Drapeaux communs

Drapeau	brève description
`re.IGNORECASE` , `re.I`	Fait le motif ignorer le cas
`re.DOTALL` , `re.S`	Fait `.` correspondre à tout, y compris les nouvelles lignes
`re.MULTILINE` , `re.M`	Donne `^` correspond au début d'une ligne et `$` la fin d'une ligne
`re.DEBUG`	Active les informations de débogage

Pour la liste complète de tous les drapeaux disponibles, vérifiez les documents

Drapeaux en ligne

De la documentation :

```
(?iLmsux)
```

(Une ou plusieurs lettres de l'ensemble 'i', 'L', 'm', 's', 'u', 'x'.)

Le groupe correspond à la chaîne vide; les lettres définissent les drapeaux correspondants: re.I (ignore case), re.L (dépendant de la locale), re.M (multi-line), re.S (dot match all), re.U (dépendant d'Unicode), et re.X (verbose), pour toute l'expression régulière. Ceci est utile si vous souhaitez inclure les drapeaux dans l'expression régulière, au lieu de transmettre un argument flag à la fonction re.compile ().

Notez que l'indicateur (? X) modifie la façon dont l'expression est analysée. Il doit être utilisé en premier dans la chaîne d'expression, ou après un ou plusieurs caractères d'espacement. S'il y a des caractères non blancs avant le drapeau, les résultats ne sont pas définis.

Itérer sur les correspondances en utilisant `re.finditer`

Vous pouvez utiliser `re.finditer` pour parcourir tous les résultats d'une chaîne. Cela vous donne (en comparaison avec `re.findall` des informations supplémentaires, telles que des informations sur l'emplacement de la correspondance dans la chaîne (index):

```
import re
text = 'You can try to find an ant in this string'
pattern = 'an?\w' # find 'an' either with or without a following word character

for match in re.finditer(pattern, text):
    # Start index of match (integer)
    sStart = match.start()

    # Final index of match (integer)
    sEnd = match.end()

    # Complete match (string)
    sGroup = match.group()

    # Print match
    print('Match "{}" found at: [{},{}]'.format(sGroup, sStart,sEnd))
```

Résultat:

```
Match "an" found at: [5,7]
Match "an" found at: [20,22]
Match "ant" found at: [23,26]
```

Correspond à une expression uniquement dans des emplacements spécifiques

Souvent, vous souhaitez faire correspondre une expression uniquement à *des* endroits *spécifiques* (les laissant intacts dans d'autres, c'est-à-dire). Considérons la phrase suivante:

```
An apple a day keeps the doctor away (I eat an apple everyday).
```

Ici, la "pomme" se produit deux fois, ce qui peut être résolu avec *des verbes de contrôle* appelés "

backtracking" qui sont supportés par le nouveau module `regex` . L'idée est la suivante:

```
forget_this | or this | and this as well | (but keep this)
```

Avec notre exemple Apple, ce serait:

```
import regex as re
string = "An apple a day keeps the doctor away (I eat an apple everyday)."
rx = re.compile(r'''
    \([^()]*\) (*SKIP)(*FAIL)  # match anything in parentheses and "throw it away"
    |                          # or
    apple                      # match an apple
    ''', re.VERBOSE)
apples = rx.findall(string)
print(apples)
# only one
```

Ceci ne correspond à "apple" que s'il peut être trouvé en dehors des parenthèses.

Voici comment cela fonctionne:

- En regardant de **gauche à droite** , le moteur regex consomme tout à gauche, le `(*SKIP)` agit comme une "assertion toujours vraie". Ensuite, il échoue correctement `(*FAIL)` et les backtracks.
- Maintenant, il arrive au point de `(*SKIP)` **de droite à gauche** (alias en revenant en arrière) où il est interdit d'aller plus loin vers la gauche. Au lieu de cela, on dit au moteur de jeter tout ce qui est à gauche et de sauter au point où le `(*SKIP)` été invoqué.

Lire Expressions régulières (Regex) en ligne:
https://riptutorial.com/fr/python/topic/632/expressions-regulieres--regex-

Chapitre 73: fichier temporaire NamedTemporaryFile

Paramètres

param	la description
mode	mode pour ouvrir le fichier, par défaut = w + b
effacer	Pour supprimer le fichier à la fermeture, default = True
suffixe	suffixe du nom de fichier, default = ''
préfixe	préfixe du nom de fichier, par défaut = 'tmp'
dir	dirname to place tempfile, default = None
buffsize	default = -1, (défaut du système d'exploitation utilisé)

Examples

Créer (et écrire dans un) fichier temporaire persistant connu

Vous pouvez créer des fichiers temporaires avec un nom visible sur le système de fichiers, accessible via la propriété `name` . Le fichier peut, sur les systèmes Unix, être configuré pour être supprimé à la fermeture (défini par le paramètre `delete` , la valeur par défaut est True) ou peut être rouverte ultérieurement.

Ce qui suit va créer et ouvrir un fichier temporaire nommé et écrire "Hello World!" à ce fichier. Le chemin du fichier temporaire est accessible par `name` , dans cet exemple, il est enregistré dans le `path` de la variable et imprimé pour l'utilisateur. Le fichier est ensuite ré-ouvert après la fermeture du fichier et le contenu du fichier temporaire est lu et imprimé pour l'utilisateur.

```
import tempfile

with tempfile.NamedTemporaryFile(delete=False) as t:
    t.write('Hello World!')
    path = t.name
    print path

with open(path) as t:
    print t.read()
```

Sortie:

```
/tmp/tmp6pireJ
```

```
Hello World!
```

Lire fichier temporaire NamedTemporaryFile en ligne:
https://riptutorial.com/fr/python/topic/3666/fichier-temporaire-namedtemporaryfile

Chapitre 74: Fichiers de décompression

Introduction

Pour extraire ou décompresser un fichier tarball, ZIP ou gzip, les modules tarfile, zipfile et gzip de Python sont fournis respectivement. Le module `TarFile.extractall(path=".", members=None)` Python fournit la fonction `TarFile.extractall(path=".", members=None)` pour extraire d'un fichier d'archive. Le module zipfile de Python fournit la fonction `ZipFile.extractall([path[, members[, pwd]]])` pour extraire ou décompresser des fichiers compressés ZIP. Enfin, le module gzip de Python fournit la classe GzipFile pour la décompression.

Examples

Utiliser Python ZipFile.extractall () pour décompresser un fichier ZIP

```
file_unzip = 'filename.zip'
unzip = zipfile.ZipFile(file_unzip, 'r')
unzip.extractall()
unzip.close()
```

Utiliser Python TarFile.extractall () pour décompresser une archive

```
file_untar = 'filename.tar.gz'
untar = tarfile.TarFile(file_untar)
untar.extractall()
untar.close()
```

Lire Fichiers de décompression en ligne: https://riptutorial.com/fr/python/topic/9505/fichiers-de-decompression

Chapitre 75: Fichiers de données externes d'entrée, de sous-ensemble et de sortie à l'aide de Pandas

Introduction

Cette section présente le code de base pour la lecture, le sous-paramétrage et l'écriture de fichiers de données externes à l'aide de pandas.

Examples

Code de base pour importer, sous-définir et écrire des fichiers de données externes à l'aide de Pandas

```
# Print the working directory
import os
print os.getcwd()
# C:\Python27\Scripts

# Set the working directory
os.chdir('C:/Users/general1/Documents/simple Python files')
print os.getcwd()
# C:\Users\general1\Documents\simple Python files

# load pandas
import pandas as pd

# read a csv data file named 'small_dataset.csv' containing 4 lines and 3 variables
my_data = pd.read_csv("small_dataset.csv")
my_data
#     x   y   z
# 0   1   2   3
# 1   4   5   6
# 2   7   8   9
# 3  10  11  12

my_data.shape       # number of rows and columns in data set
# (4, 3)

my_data.shape[0]    # number of rows in data set
# 4

my_data.shape[1]    # number of columns in data set
# 3

# Python uses 0-based indexing.  The first row or column in a data set is located
# at position 0.  In R the first row or column in a data set is located
# at position 1.

# Select the first two rows
my_data[0:2]
```

```
#    x   y   z
#0   1   2   3
#1   4   5   6

# Select the second and third rows
my_data[1:3]
#    x  y  z
# 1  4  5  6
# 2  7  8  9

# Select the third row
my_data[2:3]
#    x   y   z
#2   7   8   9

# Select the first two elements of the first column
my_data.iloc[0:2, 0:1]
#    x
# 0  1
# 1  4

# Select the first element of the variables y and z
my_data.loc[0, ['y', 'z']]
# y    2
# z    3

# Select the first three elements of the variables y and z
my_data.loc[0:2, ['y', 'z']]
#    y  z
# 0  2  3
# 1  5  6
# 2  8  9

# Write the first three elements of the variables y and z
# to an external file.  Here index = 0 means do not write row names.

my_data2 = my_data.loc[0:2, ['y', 'z']]

my_data2.to_csv('my.output.csv', index = 0)
```

Lire Fichiers de données externes d'entrée, de sous-ensemble et de sortie à l'aide de Pandas en ligne: https://riptutorial.com/fr/python/topic/8854/fichiers-de-donnees-externes-d-entree--de-sous-ensemble-et-de-sortie-a-l-aide-de-pandas

Chapitre 76: Fichiers et dossiers E / S

Introduction

Lorsqu'il s'agit de stocker, de lire ou de communiquer des données, le travail avec les fichiers d'un système d'exploitation est à la fois nécessaire et facile avec Python. Contrairement à d'autres langages où l'entrée et la sortie de fichiers requièrent des objets complexes de lecture et d'écriture, Python simplifie le processus en n'exigeant que des commandes pour ouvrir, lire / écrire et fermer le fichier. Cette rubrique explique comment Python peut interfacer avec les fichiers du système d'exploitation.

Syntaxe

- file_object = open (filename [, access_mode] [, mise en mémoire tampon])

Paramètres

Paramètre	Détails
nom de fichier	le chemin d'accès à votre fichier ou, si le fichier se trouve dans le répertoire de travail, le nom de fichier de votre fichier
Mode d'accès	une valeur de chaîne qui détermine comment le fichier est ouvert
mise en mémoire tampon	une valeur entière utilisée pour la mise en mémoire tampon des lignes en option

Remarques

Éviter l'enfer d'encodage multiplateforme

Lorsque vous utilisez `open()` intégré à Python, il est recommandé de toujours passer l'argument d' `encoding` si vous souhaitez que votre code soit exécuté sur plusieurs plates-formes. La raison en est que l'encodage par défaut d'un système diffère d'une plateforme à l'autre.

Bien que les systèmes `linux` utilisent effectivement `utf-8` par défaut, ce n'est ***pas*** forcément vrai pour MAC et Windows.

Pour vérifier le codage par défaut d'un système, essayez ceci:

```
import sys
sys.getdefaultencoding()
```

à partir de n'importe quel interpréteur Python.

Par conséquent, il est sage de toujours spécifier un encodage pour vous assurer que les chaînes avec lesquelles vous travaillez sont codées comme vous le pensez, garantissant ainsi la compatibilité entre les plates-formes.

```
with open('somefile.txt', 'r', encoding='UTF-8') as f:
    for line in f:
        print(line)
```

Examples

Modes de fichier

Vous pouvez ouvrir un fichier avec différents modes, spécifiés par le paramètre `mode` . Ceux-ci inclus:

- `'r'` - mode de lecture. Le défaut. Cela vous permet seulement de lire le fichier, pas de le modifier. Lorsque vous utilisez ce mode, le fichier doit exister.
- `'w'` - mode d'écriture. Il créera un nouveau fichier s'il n'existe pas, sinon le fichier sera effacé et vous pourrez y écrire.
- `'a'` - mode d'ajout. Il écrira des données à la fin du fichier. Il n'efface pas le fichier et le fichier doit exister pour ce mode.
- `'rb'` - mode de lecture en binaire. Ceci est similaire à `r` sauf que la lecture est forcée en mode binaire. C'est aussi un choix par défaut.
- `'r+'` - mode de lecture plus mode d'écriture en même temps. Cela vous permet de lire et d'écrire dans des fichiers en même temps sans avoir à utiliser `r` et `w` .
- `'rb+'` - mode lecture et écriture en binaire. La même chose que `r+` sauf que les données sont en binaire
- `'wb'` - mode d’écriture en binaire. La même chose que `w` sauf que les données sont en binaire.
- `'w+'` - mode d'écriture et de lecture. Le même que `r+` mais si le fichier n'existe pas, un nouveau est créé. Sinon, le fichier est écrasé.
- `'wb+'` - mode d’écriture et de lecture en mode binaire. La même chose que `w+` mais les données sont en binaire.
- `'ab'` - ajout en mode binaire. Similaire à `a` sauf que les données sont en binaire.
- `'a+'` - mode d’ajout et de lecture. Semblable à `w+` car il créera un nouveau fichier si le fichier n'existe pas. Sinon, le pointeur de fichier se trouve à la fin du fichier s'il existe.
- `'ab+'`

- mode d'ajout et de lecture en binaire. La même chose que `a+` sauf que les données sont en binaire.

```
with open(filename, 'r') as f:
    f.read()
with open(filename, 'w') as f:
    f.write(filedata)
with open(filename, 'a') as f:
    f.write('\n' + newdata)
```

	r	r +	w	w +	une	un +
Lis	✔	✔	✘	✔	✘	✔
Écrire	✘	✔	✔	✔	✔	✔
Crée un fichier	✘	✘	✔	✔	✔	✔
Efface le fichier	✘	✘	✔	✔	✘	✘
Position initiale	Début	Début	Début	Début	Fin	Fin

Python 3 a ajouté un nouveau mode de `exclusive creation` pour que vous ne puissiez pas tronquer ou écraser accidentellement un fichier existant.

- `'x'` - ouvert pour la création exclusive, soulèvera `FileExistsError` si le fichier existe déjà
- `'xb'` - ouvert pour le mode d'écriture de création exclusif en binaire. La même chose que `x` sauf que les données sont en binaire.
- `'x+'` - mode lecture et écriture. Semblable à `w+` car il créera un nouveau fichier si le fichier n'existe pas. Sinon, `FileExistsError` .
- `'xb+'` - mode d'écriture et de lecture. La même chose que `x+` mais les données sont binaires

	X	x +
Lis	✘	✔
Écrire	✔	✔
Crée un fichier	✔	✔
Efface le fichier	✘	✘
Position initiale	Début	Début

Permettez à quelqu'un d'écrire votre code ouvert de manière plus pythonique:

Python 3.x 3.3

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
    # Your error handling goes here
```

En Python 2, vous auriez fait quelque chose comme

Python 2.x 2.0

```
import os.path
if os.path.isfile(fname):
    with open("fname", "w") as fout:
        # Work with your open file
else:
    # Your error handling goes here
```

Lecture d'un fichier ligne par ligne

La manière la plus simple d'itérer ligne par ligne sur un fichier:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

`readline()` permet un contrôle plus granulaire de l'itération ligne par ligne. L'exemple ci-dessous est équivalent à celui ci-dessus:

```
with open('myfile.txt', 'r') as fp:
    while True:
        cur_line = fp.readline()
        # If the result is an empty string
        if cur_line == '':
            # We have reached the end of the file
            break
        print(cur_line)
```

Utiliser l'itérateur for loop et readline () ensemble est considéré comme une mauvaise pratique.

Plus communément, la méthode `readlines()` est utilisée pour stocker une collection itérable des lignes du fichier:

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

Cela imprimerait ce qui suit:

> Ligne 0: bonjour
>
> Ligne 1: monde

Obtenir le contenu complet d'un fichier

La méthode préférée du fichier i / o consiste à utiliser le mot-clé `with` . Cela garantira que le descripteur de fichier est fermé une fois la lecture ou l'écriture terminée.

```
with open('myfile.txt') as in_file:
    content = in_file.read()

print(content)
```

ou, pour gérer la fermeture du fichier manuellement, vous pouvez renoncer `with` et simplement appeler `close` vous:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
in_file.close()
```

Gardez à l'esprit que, sans utiliser une instruction `with` , vous risquez de garder le fichier ouvert par inadvertance au cas où une exception inattendue se présenterait comme suit:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close()  # This will never be called
```

Ecrire dans un fichier

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

Si vous ouvrez `myfile.txt` , vous verrez que son contenu est le suivant:

> Ligne 1 Ligne 2 Ligne 3 Ligne 4

Python n'ajoute pas automatiquement les sauts de ligne, vous devez le faire manuellement:

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1\n")
    f.write("Line 2\n")
    f.write("Line 3\n")
    f.write("Line 4\n")
```

> Ligne 1
> Ligne 2
> Ligne 3
> Ligne 4

N'utilisez pas `os.linesep` comme terminateur de ligne lorsque vous écrivez des fichiers ouverts en mode texte (valeur par défaut); utilisez plutôt `\n`

Si vous voulez spécifier un encodage, vous ajoutez simplement le paramètre d' `encoding` à la fonction `open` :

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

Il est également possible d'utiliser l'instruction print pour écrire dans un fichier. La mécanique est différente dans Python 2 vs Python 3, mais le concept est le même en ce sens que vous pouvez prendre la sortie qui serait allée à l'écran et l'envoyer à un fichier à la place.

Python 3.x 3.0

```
with open('fred.txt', 'w') as outfile:
    s = "I'm Not Dead Yet!"
    print(s) # writes to stdout
    print(s, file = outfile) # writes to outfile

    #Note: it is possible to specify the file parameter AND write to the screen
    #by making sure file ends up with a None value either directly or via a variable
    myfile = None
    print(s, file = myfile) # writes to stdout
    print(s, file = None)   # writes to stdout
```

En Python 2, vous auriez fait quelque chose comme

Python 2.x 2.0

```
outfile = open('fred.txt', 'w')
s = "I'm Not Dead Yet!"
print s   # writes to stdout
print >> outfile, s   # writes to outfile
```

Contrairement à la fonction d'écriture, la fonction d'impression ajoute automatiquement des sauts de ligne.

Copier le contenu d'un fichier dans un fichier différent

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- En utilisant le module `shutil` :

```
import shutil
shutil.copyfile(src, dst)
```

Vérifiez si un fichier ou un chemin existe

Utilisez le style de codage EAFP et `try` de l'ouvrir.

```
import errno

try:
    with open(path) as f:
        # File exists
except IOError as e:
    # Raise the exception if it is not ENOENT (No such file or directory)
    if e.errno != errno.ENOENT:
        raise
    # No such file or directory
```

Cela évitera également les conditions de concurrence si un autre processus supprime le fichier entre la vérification et le moment où il est utilisé. Cette condition de concurrence peut survenir dans les cas suivants:

- En utilisant le module `os` :

```
import os
os.path.isfile('/path/to/some/file.txt')
```

Python 3.x 3.4

- En utilisant `pathlib` :

```
import pathlib
path = pathlib.Path('/path/to/some/file.txt')
if path.is_file():
    ...
```

Pour vérifier si un chemin donné existe ou non, vous pouvez suivre la procédure EAFP ci-dessus ou vérifier explicitement le chemin:

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## Do stuff
```

Copier une arborescence de répertoires

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

Le répertoire de destination **ne doit pas** déjà **exister** .

Itérer les fichiers (récursivement)

Pour itérer tous les fichiers, y compris les sous-répertoires, utilisez os.walk:

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

root_dir peut être "." pour démarrer à partir du répertoire en cours ou de tout autre chemin à partir duquel commencer.

Python 3.x 3.5

Si vous souhaitez également obtenir des informations sur le fichier, vous pouvez utiliser la méthode plus efficace os.scandir comme suit :

```
for entry in os.scandir(path):
   if not entry.name.startswith('.') and entry.is_file():
       print(entry.name)
```

Lire un fichier entre plusieurs lignes

Supposons donc que vous souhaitiez effectuer une itération uniquement entre certaines lignes spécifiques d'un fichier

Vous pouvez utiliser les `itertools` pour cela

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

Cela lira les lignes 13 à 20, car l'indexation en python commence à 0. Le numéro de ligne 1 est donc indexé à 0

Comme peut également lire certaines lignes supplémentaires en utilisant le mot clé `next()` ici.

Et lorsque vous utilisez l'objet fichier comme une itération, n'utilisez pas l'instruction `readline()` car les deux techniques de déplacement d'un fichier ne doivent pas être mélangées.

Accès aléatoire aux fichiers à l'aide de mmap

L'utilisation du module `mmap` permet à l'utilisateur d'accéder de manière aléatoire aux emplacements d'un fichier en mappant le fichier en mémoire. Ceci est une alternative à l'utilisation des opérations de fichier normales.

```
import mmap

with open('filename.ext', 'r') as fd:
    # 0: map the whole file
    mm = mmap.mmap(fd.fileno(), 0)
```

```
    # print characters at indices 5 through 10
    print mm[5:10]

    # print the line starting from mm's current position
    print mm.readline()

    # write a character to the 5th index
    mm[5] = 'a'

    # return mm's position to the beginning of the file
    mm.seek(0)

    # close the mmap object
    mm.close()
```

Remplacement du texte dans un fichier

```
import fileinput

replacements = {'Search1': 'Replace1',
                'Search2': 'Replace2'}

for line in fileinput.input('filename.txt', inplace=True):
    for search_for in replacements:
        replace_with = replacements[search_for]
        line = line.replace(search_for, replace_with)
    print(line, end='')
```

Vérifier si un fichier est vide

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

ou

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

Cependant, les deux lanceront une exception si le fichier n'existe pas. Pour éviter d'avoir à attraper une telle erreur, procédez comme suit:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

qui renverra une valeur `bool` .

Lire Fichiers et dossiers E / S en ligne: https://riptutorial.com/fr/python/topic/267/fichiers-et-dossiers-e---s

Chapitre 77: Filtre

Syntaxe

- filtre (fonction, itérable)
- itertools.ifilter (fonction, itérable)
- future_builtins.filter (fonction, itérable)
- itertools.ifilterfalse (fonction, itérable)
- itertools.filterfalse (fonction, itérable)

Paramètres

Paramètre	Détails
fonction	*callable* qui détermine la condition ou `None` puis utilise la fonction d'identité pour le filtrage (*uniquement* pour la *position*)
itérable	iterable qui sera filtré (*uniquement positionnel*)

Remarques

Dans la plupart des cas, une expression de compréhension ou de générateur est plus lisible, plus puissante et plus efficace que `filter()` ou `ifilter()` .

Examples

Utilisation de base du filtre

Pour `filter` éléments de suppression d'une séquence en fonction de certains critères:

```
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5
```

Python 2.x 2.0

```
filter(long_name, names)
# Out: ['Barney']

[name for name in names if len(name) > 5] # equivalent list comprehension
# Out: ['Barney']


from itertools import ifilter
ifilter(long_name, names)       # as generator (similar to python 3.x filter builtin)
```

```
# Out: <itertools.ifilter at 0x4197e10>
list(ifilter(long_name, names)) # equivalent to filter with lists
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x0000000003FD5D38>
```

Python 2.x 2.6

```
# Besides the options for older python 2.x versions there is a future_builtin function:
from future_builtins import filter
filter(long_name, names)       # identical to itertools.ifilter
# Out: <itertools.ifilter at 0x3eb0ba8>
```

Python 3.x 3.0

```
filter(long_name, names)        # returns a generator
# Out: <filter at 0x1fc6e443470>
list(filter(long_name, names))  # cast to list
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x000001C6F49BF4C0>
```

Filtre sans fonction

Si le paramètre de la fonction est `None` , la fonction d'identité sera utilisée:

```
list(filter(None, [1, 0, 2, [], '', 'a']))  # discards 0, [] and ''
# Out: [1, 2, 'a']
```

Python 2.x 2.0.1

```
[i for i in [1, 0, 2, [], '', 'a'] if i] # equivalent list comprehension
```

Python 3.x 3.0.0

```
(i for i in [1, 0, 2, [], '', 'a'] if i) # equivalent generator expression
```

Filtrer comme vérification de court-circuit

`filter` (python 3.x) et `ifilter` (python 2.x) renvoient un générateur afin qu'ils puissent être très utiles lors de la création d'un test de court-circuit comme `or` ou `and` :

Python 2.x 2.0.1

```
 # not recommended in real use but keeps the example short:
from itertools import ifilter as filter
```

Python 2.x 2.6.1

```
 from future_builtins import filter
```

Pour trouver le premier élément inférieur à 100:

```
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filter(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
#        Check rectangular tire, 80$
# Out: ('rectangular tire', 80)
```

La fonction `next` donne l'élément suivant (dans ce cas en premier lieu) et est donc la raison pour laquelle il est court-circuité.

Fonction complémentaire: filterfalse, ifilterfalse

Il y a une fonction complémentaire pour le `filter` dans le `itertools` :

Python 2.x 2.0.1

```
 # not recommended in real use but keeps the example valid for python 2.x and python 3.x
from itertools import ifilterfalse as filterfalse
```

Python 3.x 3.0.0

```
from itertools import filterfalse
```

qui fonctionne exactement comme le `filter` du *générateur* mais ne garde que les éléments qui sont `False` :

```
# Usage without function (None):
list(filterfalse(None, [1, 0, 2, [], '', 'a']))  # discards 1, 2, 'a'
# Out: [0, [], '']
```

```
# Usage with function
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5

list(filterfalse(long_name, names))
# Out: ['Fred', 'Wilma']
```

```
# Short-circuit useage with next:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filterfalse(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
# Out: ('Toyota', 1000)
```

```
# Using an equivalent generator:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
generator = (car for car in car_shop if not car[1] < 100)
next(generator)
```

Lire Filtre en ligne: https://riptutorial.com/fr/python/topic/201/filtre

Chapitre 78: Fonction de la carte

Syntaxe

- map (fonction, iterable [, * additional_iterables])
- future_builtins.map (fonction, iterable [, * additional_iterables])
- itertools.imap (function, iterable [, * additional_iterables])

Paramètres

Paramètre	Détails
fonction	fonction de mappage (doit prendre autant de paramètres qu'il y a d'itérables) (*positionnel uniquement*)
itérable	la fonction est appliquée à chaque élément de l'itérable (*positionnel uniquement*)
* additional_iterables	voir itérable, mais autant que vous le souhaitez (*optionnel* , *uniquement positionnel*)

Remarques

Tout ce qui peut être fait avec la `map` peut également être fait avec des `comprehensions` :

```
list(map(abs, [-1,-2,-3]))     # [1, 2, 3]
[abs(i) for i in [-1,-2,-3]]   # [1, 2, 3]
```

Bien que vous ayez besoin de `zip` si vous avez plusieurs itérables:

```
import operator
alist = [1,2,3]
list(map(operator.add, alist, alist))  # [2, 4, 6]
[i + j for i, j in zip(alist, alist)]  # [2, 4, 6]
```

Les compréhensions de liste sont efficaces et peuvent être plus rapides que la `map` dans de nombreux cas, alors testez les temps des deux approches si la vitesse est importante pour vous.

Examples

Utilisation basique de map, itertools.imap et future_builtins.map

La fonction de carte est la plus simple parmi les éléments intégrés Python utilisés pour la programmation fonctionnelle. `map()` applique une fonction spécifiée à chaque élément dans une

itération:

```
names = ['Fred', 'Wilma', 'Barney']
```

Python 3.x 3.0

```
map(len, names)  # map in Python 3.x is a class; its instances are iterable
# Out: <map object at 0x00000198B32E2CF8>
```

Une `map` compatible avec Python 3 est incluse dans le module `future_builtins` :

Python 2.x 2.6

```
from future_builtins import map  # contains a Python 3.x compatible map()
map(len, names)                  # see below
# Out: <itertools.imap instance at 0x3eb0a20>
```

Alternativement, en Python 2, on peut utiliser `imap` d' `itertools` pour obtenir un générateur.

Python 2.x 2.3

```
map(len, names)   # map() returns a list
# Out: [4, 5, 6]

from itertools import imap
imap(len, names)  # itertools.imap() returns a generator
# Out: <itertools.imap at 0x405ea20>
```

Le résultat peut être explicitement converti en une `list` pour supprimer les différences entre Python 2 et 3:

```
list(map(len, names))
# Out: [4, 5, 6]
```

`map()` peut être remplacé par une *expression de compréhension de liste* ou de *générateur* équivalente:

```
[len(item) for item in names] # equivalent to Python 2.x map()
# Out: [4, 5, 6]

(len(item) for item in names) # equivalent to Python 3.x map()
# Out: <generator object <genexpr> at 0x00000195888D5FC0>
```

Mapper chaque valeur dans une itération

Par exemple, vous pouvez prendre la valeur absolue de chaque élément:

```
list(map(abs, (1, -1, 2, -2, 3, -3))) # the call to `list` is unnecessary in 2.x
# Out: [1, 1, 2, 2, 3, 3]
```

La fonction anonyme prend également en charge le mappage d'une liste:

```
map(lambda x:x*2, [1, 2, 3, 4, 5])
# Out: [2, 4, 6, 8, 10]
```

ou convertir les valeurs décimales en pourcentages:

```
def to_percent(num):
    return num * 100

list(map(to_percent, [0.95, 0.75, 1.01, 0.1]))
# Out: [95.0, 75.0, 101.0, 10.0]
```

ou convertir des dollars en euros (compte tenu d'un taux de change):

```
from functools import partial
from operator import mul

rate = 0.9  # fictitious exchange rate, 1 dollar = 0.9 euros
dollars = {'under_my_bed': 1000,
           'jeans': 45,
           'bank': 5000}

sum(map(partial(mul, rate), dollars.values()))
# Out: 5440.5
```

`functools.partial` est un moyen pratique de corriger les paramètres des fonctions pour qu'elles puissent être utilisées avec `map` au lieu d'utiliser `lambda` ou de créer des fonctions personnalisées.

Mappage des valeurs de différentes itérations

Par exemple le calcul de la moyenne de chaque `i` -ième élément de multiples iterables:

```
def average(*args):
    return float(sum(args)) / len(args)  # cast to float - only mandatory for python 2.x

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117, 91, 102]
measurement3 = [104, 102, 95, 101]

list(map(average, measurement1, measurement2, measurement3))
# Out: [102.0, 110.0, 95.0, 100.0]
```

Il existe différentes exigences si plusieurs itérables sont transmises à `map` selon la version de python:

- La fonction doit prendre autant de paramètres qu'il ya d'itérables:

  ```
  def median_of_three(a, b, c):
      return sorted((a, b, c))[1]

  list(map(median_of_three, measurement1, measurement2))
  ```

  TypeError: median_of_three () manquant 1 argument de position requis: 'c'

```
list(map(median_of_three, measurement1, measurement2, measurement3, measurement3))
```

TypeError: median_of_three () prend 3 arguments positionnels mais 4 ont été donnés

Python 2.x 2.0.1

- `map` : Les itère de cartographie aussi longtemps que l' un itérable est pas encore totalement consommé , mais suppose `None` des iterables entièrement consommés:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
```

TypeError: type (s) d'opérande non pris en charge pour -: 'int' et 'NoneType'

- `itertools.imap` et `future_builtins.map` : le mappage s'arrête dès qu'un arrête est itérable:

```
import operator
from itertools import imap

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(imap(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(imap(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Python 3.x 3.0.0

- Le mappage s’arrête dès qu’un arret itable:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(map(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Transposer avec Map: Utiliser "None" comme argument de fonction (python 2.x uniquement)

```
from itertools import imap
```

```
from future_builtins import map as fmap # Different name to highlight differences

image = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]

list(map(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(fmap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(imap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

image2 = [[1, 2, 3],
          [4, 5],
          [7, 8, 9]]
list(map(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8), (3, None, 9)]  # Fill missing values with None
list(fmap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # ignore columns with missing values
list(imap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # dito
```

Python 3.x 3.0.0

```
list(map(None, *image))
```

TypeError: l'objet 'NoneType' n'est pas appelable

Mais il existe une solution pour avoir des résultats similaires:

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# Out: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Cartographie en série et parallèle

map () est une fonction intégrée, ce qui signifie qu'elle est disponible partout sans avoir à utiliser une instruction «importation». Il est disponible partout, tout comme print () Si vous regardez l'exemple 5, vous verrez que je devais utiliser une instruction d'importation avant de pouvoir utiliser une jolie impression (importation pprint). Ainsi, pprint n'est pas une fonction intégrée

Cartographie en série

Dans ce cas, chaque argument de l'itérable est fourni comme argument de la fonction de mappage dans l'ordre croissant. Cela se produit lorsque nous n'avons qu'une seule option à mapper et que la fonction de mappage nécessite un seul argument.

Exemple 1

```
insects = ['fly', 'ant', 'beetle', 'cankerworm']
f = lambda x: x + ' is an insect'
```

```
print(list(map(f, insects))) # the function defined by f is executed on each item of the
iterable insects
```

résulte en

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

Exemple 2

```
print(list(map(len, insects))) # the len function is executed each item in the insect list
```

résulte en

```
[3, 3, 6, 10]
```

Cartographie parallèle

Dans ce cas, chaque argument de la fonction de mappage est extrait de toutes les itérables (une de chaque itérable) en parallèle. Le nombre d'itérables fourni doit donc correspondre au nombre d'arguments requis par la fonction.

```
carnivores = ['lion', 'tiger', 'leopard', 'arctic fox']
herbivores = ['african buffalo', 'moose', 'okapi', 'parakeet']
omnivores = ['chicken', 'dove', 'mouse', 'pig']

def animals(w, x, y, z):
    return '{0}, {1}, {2}, and {3} ARE ALL ANIMALS'.format(w.title(), x, y, z)
```

Exemple 3

```
# Too many arguments
# observe here that map is trying to pass one item each from each of the four iterables to
len. This leads len to complain that
# it is being fed too many arguments
print(list(map(len, insects, carnivores, herbivores, omnivores)))
```

résulte en

```
TypeError: len() takes exactly one argument (4 given)
```

Exemple 4

```
# Too few arguments
# observe here that map is suppose to execute animal on individual elements of insects one-by-
one. But animals complain when
# it only gets one argument, whereas it was expecting four.
print(list(map(animals, insects)))
```

résulte en

```
TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Exemple 5

```
# here map supplies w, x, y, z with one value from across the list
import pprint
pprint.pprint(list(map(animals, insects, carnivores, herbivores, omnivores)))
```

résulte en

```
 ['Fly, lion, african buffalo, and chicken ARE ALL ANIMALS',
 'Ant, tiger, moose, and dove ARE ALL ANIMALS',
 'Beetle, leopard, okapi, and mouse ARE ALL ANIMALS',
 'Cankerworm, arctic fox, parakeet, and pig ARE ALL ANIMALS']
```

Lire Fonction de la carte en ligne: https://riptutorial.com/fr/python/topic/333/fonction-de-la-carte

Chapitre 79: Fonctions partielles

Introduction

Comme vous le savez probablement si vous venez de l'école de POO, la spécialisation d'une classe abstraite et son utilisation est une pratique que vous devez garder à l'esprit lorsque vous écrivez votre code.

Et si vous pouviez définir une fonction abstraite et la spécialiser pour en créer différentes versions? Le considère comme une sorte de *fonction Héritage* où vous liez des paramètres spécifiques pour les rendre fiables pour un scénario spécifique.

Syntaxe

- partial (fonction, ** params_you_want_fix)

Paramètres

Param	détails
X	le nombre à soulever
y	l'exposant
élever	la fonction à être spécialisée

Remarques

Comme indiqué dans Python doc les *functools.partial* :

> Renvoie un nouvel objet partiel qui, lorsqu'il sera appelé, se comportera comme func appelé avec les arguments positionnels args et les mots clés arguments. Si plusieurs arguments sont fournis à l'appel, ils sont ajoutés aux arguments. Si des arguments de mots-clés supplémentaires sont fournis, ils étendent et remplacent les mots-clés.

Consultez ce lien pour voir comment une *partie* peut être mise en œuvre.

Examples

Élever le pouvoir

Supposons que nous voulions augmenter *x* à un nombre *y* .

Vous écrivez ceci comme:

```
def raise_power(x, y):
    return x**y
```

Que se passe-t-il si votre valeur *y* peut prendre un ensemble fini de valeurs?

Supposons que *y* soit un de *[3,4,5]* et supposons que vous ne souhaitiez pas offrir à l'utilisateur final la possibilité d'utiliser une telle fonction, car elle nécessite beaucoup de calculs. En fait, vous devriez vérifier si *y a* une valeur valide et réécrire votre fonction en tant que:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Désordonné? Utilisons la forme abstraite et spécialisons-la dans les trois cas: implémentons-les **partiellement** .

```
from functors import partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

Que se passe t-il ici? Nous avons fixé les paramètres *y* et défini trois fonctions différentes.

Nul besoin d'utiliser la fonction abstraite définie ci-dessus (vous pourriez la rendre *privée*), mais vous pourriez utiliser **des** fonctions **partielles appliquées** pour traiter un nombre à une valeur fixe.

Lire Fonctions partielles en ligne: https://riptutorial.com/fr/python/topic/9383/fonctions-partielles

Chapitre 80: Formatage de chaîne

Introduction

Lors du stockage et de la transformation des données pour les humains, le formatage des chaînes peut devenir très important. Python offre une grande variété de méthodes de formatage de chaînes décrites dans cette rubrique.

Syntaxe

- "{}". format (42) ==> "42"
- "{0}". Format (42) ==> "42"
- "{0: .2f}". Format (42) ==> "42.00"
- "{0: .0f}". Format (42.1234) ==> "42"
- "{réponse}". format (no_answer = 41, réponse = 42) ==> "42"
- "{answer: .2f}". format (no_answer = 41, réponse = 42) ==> "42.00"
- "{[clé]}". format ({'clé': 'valeur'}) ==> "valeur"
- "{[1]}". Format (['zéro', 'un', 'deux']) ==> "un"
- "{réponse} = {réponse}". format (réponse = 42) ==> "42 = 42"
- " .join (['stack', 'overflow']) ==> "débordement de pile"

Remarques

- Devrait vérifier PyFormat.info pour une introduction / explication très approfondie et douce de son fonctionnement.

Examples

Bases du formatage de chaînes

```
foo = 1
bar = 'bar'
baz = 3.14
```

Vous pouvez utiliser `str.format` pour formater la sortie. Les paires de crochets sont remplacées par des arguments dans l'ordre dans lequel les arguments sont transmis:

```
print('{}, {} and {}'.format(foo, bar, baz))
# Out: "1, bar and 3.14"
```

Les index peuvent également être spécifiés entre crochets. Les nombres correspondent aux index des arguments passés à la fonction `str.format` (basée sur 0).

```
print('{0}, {1}, {2}, and {1}'.format(foo, bar, baz))
```

```
# Out: "1, bar, 3.14, and bar"
print('{0}, {1}, {2}, and {3}'.format(foo, bar, baz))
# Out: index out of range error
```

Les arguments nommés peuvent également être utilisés:

```
print("X value is: {x_val}. Y value is: {y_val}.".format(x_val=2, y_val=3))
# Out: "X value is: 2. Y value is: 3."
```

Les attributs d'objet peuvent être référencés lorsqu'ils sont passés dans `str.format` :

```
class AssignValue(object):
    def __init__(self, value):
        self.value = value
my_value = AssignValue(6)
print('My value is: {0.value}'.format(my_value))  # "0" is optional
# Out: "My value is: 6"
```

Les clés de dictionnaire peuvent également être utilisées:

```
my_dict = {'key': 6, 'other_key': 7}
print("My other key is: {0[other_key]}".format(my_dict))  # "0" is optional
# Out: "My other key is: 7"
```

La même chose s'applique aux index de liste et de tuple:

```
my_list = ['zero', 'one', 'two']
print("2nd element is: {0[2]}".format(my_list))  # "0" is optional
# Out: "2nd element is: two"
```

> Remarque: en plus de `str.format` , Python fournit également l'opérateur modulo `%` également connu sous le nom d' *opérateur de formatage* ou d' *interpolation de chaîne* (voir PEP 3101) - pour le formatage des chaînes. `str.format` est un successeur de `%` et offre une plus grande flexibilité, par exemple en facilitant l'exécution de multiples substitutions.

Outre les index d'argument, vous pouvez également inclure une *spécification de format* dans les accolades. Ceci est une expression qui suit des règles particulières et doit être précédée par deux points (`:`). Consultez la documentation pour une description complète de la spécification de format. Un exemple de spécification de format est la directive d'alignement `:~^20` (`^` représente l'alignement central, largeur totale 20, remplissage avec ~ caractère):

```
'{:~^20}'.format('centered')
# Out: '~~~~~~centered~~~~~~'
```

`format` permet un comportement impossible avec `%` , par exemple la répétition des arguments:

```
t = (12, 45, 22222, 103, 6)
print '{0} {2} {1} {2} {3} {2} {4} {2}'.format(*t)
# Out: 12 22222 45 22222 103 22222 6 22222
```

Le `format` étant une fonction, il peut être utilisé comme argument dans d'autres fonctions:

```
number_list = [12,45,78]
print map('the number is {}'.format, number_list)
# Out: ['the number is 12', 'the number is 45', 'the number is 78']


from datetime import datetime,timedelta

once_upon_a_time = datetime(2010, 7, 1, 12, 0, 0)
delta = timedelta(days=13, hours=8,  minutes=20)

gen = (once_upon_a_time + x * delta for x in xrange(5))

print '\n'.join(map('{:%Y-%m-%d %H:%M:%S}'.format, gen))
#Out: 2010-07-01 12:00:00
#     2010-07-14 20:20:00
#     2010-07-28 04:40:00
#     2010-08-10 13:00:00
#     2010-08-23 21:20:00
```

Alignement et remplissage

Python 2.x 2.6

La méthode `format()` peut être utilisée pour modifier l'alignement de la chaîne. Vous devez le faire avec une expression de format de la forme `:[fill_char][align_operator][width]` où `align_operator` est l'un des suivants:

- `<` force le champ à être aligné à gauche dans la `width` .
- `>` force le champ à être aligné à droite dans la `width` .
- `^` force le champ à être centré dans la `width` .
- `=` force le remplissage à être placé après le signe (types numériques uniquement).

`fill_char` (si omis par défaut est un espace) est le caractère utilisé pour le remplissage.

```
'{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

'{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'

'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'
```

Remarque: vous pouvez obtenir les mêmes résultats en utilisant les fonctions de chaîne `ljust()` , `rjust()` , `center()` , `zfill()` , mais ces fonctions sont obsolètes depuis la version 2.5.

Format littéraux (f-string)

Chaînes de format littérales ont été introduits dans PEP 498 (Python3.6 et vers le haut), vous

permettant de préfixer `f` au début d'une chaîne littérale pour appliquer efficacement `.format` à avec toutes les variables du périmètre actuel.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

Cela fonctionne avec des chaînes de format plus avancées, y compris l'alignement et la notation par points.

```
>>> f'{foo:^7s}'
'  bar  '
```

Note: Le `f''` ne désigne pas un type particulier comme `b''` pour les `bytes` ou `u''` pour `unicode` dans python2. Le formage est immédiatement appliqué, entraînant un brassage normal.

Les chaînes de format peuvent également être *imbriquées* :

```
>>> price = 478.23
>>> f"{f'${price:0.2f}':*>20s}"
'*************$478.23'
```

Les expressions d'une chaîne de caractères sont évaluées dans l'ordre de gauche à droite. Ceci est détectable uniquement si les expressions ont des effets secondaires:

```
>>> def fn(l, incr):
...     result = l[0]
...     l[0] += incr
...     return result
...
>>> lst = [0]
>>> f'{fn(lst,2)} {fn(lst,3)}'
'0 2'
>>> f'{fn(lst,2)} {fn(lst,3)}'
'5 7'
>>> lst
[10]
```

Formatage de chaîne avec datetime

Toute classe peut configurer sa propre syntaxe de formatage de chaîne via la méthode `__format__` . Un type dans la bibliothèque Python standard qui en fait une utilisation pratique est le type `datetime` , où l'on peut utiliser des codes de mise en forme semblables à `strftime` directement dans `str.format` :

```
>>> from datetime import datetime
>>> 'North America: {dt:%m/%d/%Y}.  ISO: {dt:%Y-%m-%d}.'.format(dt=datetime.now())
'North America: 07/21/2016.  ISO: 2016-07-21.'
```

Une liste complète de la liste des formateurs datetime peut être trouvée dans la documentation officielle .

Format utilisant Getitem et Getattr

Toute structure de données qui prend en charge `__getitem__` peut avoir sa structure imbriquée formatée:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

Les attributs d'objet sont accessibles via `getattr()` :

```
class Person(object):
    first = 'Zaphod'
    last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

Formatage flottant

```
>>> '{0:.0f}'.format(42.12345)
'42'

>>> '{0:.1f}'.format(42.12345)
'42.1'

>>> '{0:.3f}'.format(42.12345)
'42.123'

>>> '{0:.5f}'.format(42.12345)
'42.12345'

>>> '{0:.7f}'.format(42.12345)
'42.1234500'
```

Même attente pour une autre manière de référencer:

```
>>> '{:.3f}'.format(42.12345)
'42.123'

>>> '{answer:.3f}'.format(answer=42.12345)
'42.123'
```

Les nombres à virgule flottante peuvent également être formatés en notation scientifique ou en pourcentage:

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'

>>> '{0:.0%}'.format(42.12345)
'4212%'
```

Vous pouvez également combiner les notations `{0}` et `{name}` . Ceci est particulièrement utile

lorsque vous souhaitez arrondir toutes les variables à un nombre prédéfini de décimales *avec une déclaration* :

```
>>> s = 'Hello'
>>> a, b, c = 1.12345, 2.34567, 34.5678
>>> digits = 2

>>> '{0}! {1:.{n}f}, {2:.{n}f}, {3:.{n}f}'.format(s, a, b, c, n=digits)
'Hello! 1.12, 2.35, 34.57'
```

Formatage des valeurs numériques

La méthode `.format()` peut interpréter un nombre sous différents formats, tels que:

```
>>> '{:c}'.format(65)    # Unicode character
'A'

>>> '{:d}'.format(0x0a)  # base 10
'10'

>>> '{:n}'.format(0x0a)  # base 10 using current locale for separators
'10'
```

Formater les entiers en différentes bases (hexadécimal, oct, binaire)

```
>>> '{0:x}'.format(10) # base 16, lowercase – Hexadecimal
'a'

>>> '{0:X}'.format(10) # base 16, uppercase – Hexadecimal
'A'

>>> '{:o}'.format(10) # base 8 – Octal
'12'

>>> '{:b}'.format(10) # base 2 – Binary
'1010'

>>> '{0:#b}, {0:#o}, {0:#x}'.format(42) # With prefix
'0b101010, 0o52, 0x2a'

>>> '8 bit: {0:08b}; Three bytes: {0:06x}'.format(42) # Add zero padding
'8 bit: 00101010; Three bytes: 00002a'
```

Utilisez le formatage pour convertir un tuple flottant RVB en une chaîne hexadécimale couleur:

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{:02X}{:02X}{:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'#FF6600'
```

Seuls les entiers peuvent être convertis:

```
>>> '{:x}'.format(42.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ValueError: Unknown format code 'x' for object of type 'float'
```

Formatage personnalisé pour une classe

Remarque:

> Tout ce qui suit s'applique à la méthode `str.format` , ainsi qu'à la fonction de `format` . Dans le texte ci-dessous, les deux sont interchangeables.

Pour chaque valeur transmise à la fonction `format` , Python recherche une méthode `__format__` pour cet argument. Votre propre classe personnalisée peut donc avoir sa propre méthode `__format__` pour déterminer comment la fonction de `format` affichera et formatera votre classe et ses attributs.

Ceci est différent de la méthode `__str__` , car dans la méthode `__format__` , vous pouvez prendre en compte le langage de formatage, y compris l'alignement, la largeur des champs, et même (si vous le souhaitez) implémenter vos propres spécificateurs de format et vos propres extensions de langage de formatage. 1

```
object.__format__(self, format_spec)
```

Par exemple :

```
# Example in Python 2 - but can be easily applied to Python 3

class Example(object):
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a,b,c

    def __format__(self, format_spec):
        """ Implement special semantics for the 's' format specifier """
        # Reject anything that isn't an s
        if format_spec[-1] != 's':
            raise ValueError('{} format specifier not understood for this object',
format_spec[:-1])

        # Output in this example will be (<a>,<b>,<c>)
        raw = "(" + ",".join([str(self.a), str(self.b), str(self.c)]) + ")"
        # Honor the format language by using the inbuilt string format
        # Since we know the original format_spec ends in an 's'
        # we can take advantage of the str.format method with a
        # string argument we constructed above
        return "{r:{f}}".format( r=raw, f=format_spec )

inst = Example(1,2,3)
print "{0:>20s}".format( inst )
# out :              (1,2,3)
# Note how the right align and field width of 20 has been honored.
```

Remarque:

> Si votre classe personnalisée n'a pas une coutume `__format__` méthode et une instance de la classe est passée à la `format` fonction, **python2** utilisera toujours la valeur de

retour de la `__str__` méthode ou `__repr__` méthode pour déterminer ce qu'il faut imprimer (et si elles existent ni alors la default `repr` sera utilisé), et vous devrez utiliser le spécificateur de format `s` pour le formater. Avec **Python3** , pour passer votre classe personnalisée à la fonction `format` , vous aurez besoin de définir la méthode `__format__` sur votre classe personnalisée.

Format imbriqué

Certains formats peuvent prendre des paramètres supplémentaires, tels que la largeur de la chaîne formatée ou l'alignement:

```
>>> '{:.>10}'.format('foo')
'.......foo'
```

Ceux-ci peuvent également être fournis en tant que paramètres à `format` en imbriquant plus `{}` dans le `{}` :

```
>>> '{:.>{}}'.format('foo', 10)
'.......foo'
'{:{}{}{}}'.format('foo', '*', '^', 15)
'******foo******'
```

Dans ce dernier exemple, la chaîne de format `'{:{}{}{}}'` est modifiée en `'{:*^15}'` (c.-à-d. Centre et pavé avec * pour une longueur totale de 15 ") avant de l'appliquer au chaîne réelle `'foo'` à formater de cette façon.

Cela peut être utile dans les cas où les paramètres ne sont pas connus à l'avance, pour les instances lors de l'alignement des données tabulaires:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
bbbbbbb
    ccc
```

Cordes de rembourrage et de troncature, combinées

Supposons que vous souhaitiez imprimer des variables dans une colonne de 3 caractères.

Note: doubler `{` et `}` leur échappe.

```
s = """

pad
{{:3}}             :{a:3}:

truncate
{{:.3}}            :{e:.3}:
```

```
combined
{{:>3.3}}          :{a:>3.3}:
{{:3.3}}           :{a:3.3}:
{{:3.3}}           :{c:3.3}:
{{:3.3}}           :{e:3.3}:
"""

print (s.format(a="1"*1, c="3"*3, e="5"*5))
```

Sortie:

```
pad
{:3}             :1  :

truncate
{:.3}            :555:

combined
{:>3.3}          :  1:
{:3.3}           :1  :
{:3.3}           :333:
{:3.3}           :555:
```

Espaces réservés nommés

Les chaînes de format peuvent contenir des marqueurs nommés qui sont interpolées en utilisant des arguments de mots clés pour le `format` .

Utiliser un dictionnaire (Python 2.x)

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

Utiliser un dictionnaire (Python 3.2+)

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

`str.format_map` permet d'utiliser des dictionnaires sans avoir à les déballer en premier. De plus, la classe de `data` (qui peut être un type personnalisé) est utilisée à la place d'un `dict` nouvellement rempli.

Sans dictionnaire:

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
'Hodor Hodor!'
```

Lire Formatage de chaîne en ligne: https://riptutorial.com/fr/python/topic/1019/formatage-de-chaine

Chapitre 81: Formatage de date

Examples

Temps entre deux dates

```
from datetime import datetime

a = datetime(2016,10,06,0,0,0)
b = datetime(2016,10,01,23,59,59)

a-b
# datetime.timedelta(4, 1)

(a-b).days
# 4
(a-b).total_seconds()
# 518399.0
```

Chaîne d'analyse vers l'objet datetime

Utilise les codes de format standard C.

```
from datetime import datetime
datetime_string = 'Oct 1 2016, 00:00:00'
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strptime(datetime_string, datetime_string_format)
# datetime.datetime(2016, 10, 1, 0, 0)
```

Sortie de l'objet datetime en chaîne

Utilise les codes de format standard C.

```
from datetime import datetime
datetime_for_string = datetime(2016,10,1,0,0)
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strftime(datetime_for_string,datetime_string_format)
# Oct 01 2016, 00:00:00
```

Lire Formatage de date en ligne: https://riptutorial.com/fr/python/topic/7284/formatage-de-date

Chapitre 82: Générateurs

Introduction

Les générateurs sont des itérateurs paresseux créés par des fonctions de générateur (à l'aide de `yield`) ou des expressions de générateur (using `(an_expression for x in an_iterator)`).

Syntaxe

- rendement `<expr>`
- rendement de `<expr>`
- `<var>` = rendement `<expr>`
- suivant (`<iter>`)

Examples

Itération

Un objet générateur prend en charge le *protocole itérateur* . En d'autres `__next__()` , il fournit une méthode `next()` (`__next__()` dans Python 3.x), qui est utilisée pour `__iter__` son exécution, et sa méthode `__iter__` . Cela signifie qu'un générateur peut être utilisé dans n'importe quelle construction de langage prenant en charge les objets itérables génériques.

```
# naive partial implementation of the Python 2.x xrange()
def xrange(n):
    i = 0
    while i < n:
        yield i
        i += 1

# looping
for i in xrange(10):
    print(i)  # prints the values 0, 1, ..., 9

# unpacking
a, b, c = xrange(3)  # 0, 1, 2

# building a list
l = list(xrange(10))  # [0, 1, ..., 9]
```

La fonction next ()

La fonction intégrée `next()` est un wrapper pratique qui peut être utilisé pour recevoir une valeur de n'importe quel itérateur (y compris un générateur d'itérateurs) et pour fournir une valeur par défaut si l'itérateur est épuisé.

```
def nums():
```

```
    yield 1
    yield 2
    yield 3
generator = nums()

next(generator, None)  # 1
next(generator, None)  # 2
next(generator, None)  # 3
next(generator, None)  # None
next(generator, None)  # None
# ...
```

La syntaxe est la `next(iterator[, default])` . Si l'itérateur se termine et qu'une valeur par défaut a été transmise, il est renvoyé. Si aucune valeur par défaut n'a été fournie, `StopIteration` est `StopIteration` .

Envoi d'objets à un générateur

En plus de recevoir des valeurs d'un générateur, il est possible d' *envoyer* un objet à un générateur à l'aide de la méthode `send()` .

```
def accumulator():
    total = 0
    value = None
    while True:
        # receive sent value
        value = yield total
        if value is None: break
        # aggregate values
        total += value

generator = accumulator()

# advance until the first "yield"
next(generator)      # 0

# from this point on, the generator aggregates values
generator.send(1)    # 1
generator.send(10)   # 11
generator.send(100)  # 111
# ...

# Calling next(generator) is equivalent to calling generator.send(None)
next(generator)      # StopIteration
```

Ce qui se passe ici est le suivant:

- Lorsque vous appelez `next(generator)` , le programme avance à la première déclaration de `yield` et retourne la valeur de `total` à ce point, qui est 0. L'exécution du générateur est suspendue à ce stade.
- Lorsque vous appelez ensuite `generator.send(x)` , l'interpréteur prend l'argument `x` et en fait la valeur de retour de la dernière instruction de `yield` , qui est affectée à la `value` . Le générateur se déroule alors comme d'habitude, jusqu'à ce qu'il produise la valeur suivante.
- Lorsque vous appelez enfin `next(generator)` , le programme traite cela comme si vous

envoyiez `None` au générateur. Il n'y a rien de particulier à propos de `None` , cependant, cet exemple utilise `None` comme valeur spéciale pour demander au générateur de s'arrêter.

Expressions du générateur

Il est possible de créer des itérateurs de générateur en utilisant une syntaxe de type compréhension.

```
generator = (i * 2 for i in range(3))

next(generator)  # 0
next(generator)  # 2
next(generator)  # 4
next(generator)  # raises StopIteration
```

Si une fonction n'a pas nécessairement besoin d'une liste, vous pouvez enregistrer des caractères (et améliorer la lisibilité) en plaçant une expression de générateur dans un appel de fonction. La parenthèse de l'appel de fonction fait implicitement de votre expression une expression de générateur.

```
sum(i ** 2 for i in range(4))  # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

De plus, vous économiserez de la mémoire car au lieu de charger la liste complète que vous parcourez (`[0, 1, 2, 3]` dans l'exemple ci-dessus), le générateur permet à Python d'utiliser les valeurs nécessaires.

introduction

Les expressions de générateur sont similaires à celles de liste, dictionnaire et ensemble, mais sont entourées de parenthèses. Les parenthèses ne doivent pas nécessairement être présentes lorsqu'elles sont utilisées comme seul argument pour un appel de fonction.

```
expression = (x**2 for x in range(10))
```

Cet exemple génère les 10 premiers carrés parfaits, dont 0 (dans lequel x = 0).

Les fonctions du générateur sont similaires aux fonctions normales, sauf qu'elles comportent un ou plusieurs énoncés de `yield` . Ces fonctions ne peuvent `return` aucune valeur (toutefois, les `return` vides sont autorisés si vous souhaitez arrêter le générateur au début).

```
def function():
    for x in range(10):
        yield x**2
```

Cette fonction de générateur est équivalente à l'expression précédente du générateur, elle produit la même chose.

Note : toutes les expressions du générateur ont leurs propres fonctions *équivalentes* , mais pas l'inverse.

Une expression de générateur peut être utilisée sans parenthèses si les deux parenthèses sont répétées autrement:

```
sum(i for i in range(10) if i % 2 == 0)   #Output: 20
any(x = 0 for x in foo)                   #Output: True or False depending on foo
type(a > b for a in foo if a % 2 == 1)    #Output: <class 'generator'>
```

Au lieu de:

```
sum((i for i in range(10) if i % 2 == 0))
any((x = 0 for x in foo))
type((a > b for a in foo if a % 2 == 1))
```

Mais non:

```
fooFunction(i for i in range(10) if i % 2 == 0,foo,bar)
return x = 0 for x in foo
barFunction(baz, a > b for a in foo if a % 2 == 1)
```

L'appel d'une fonction de générateur produit un **objet générateur** , qui peut ensuite être itéré. Contrairement aux autres types d'itérateurs, les objets générateurs ne peuvent être parcourus qu'une seule fois.

```
g1 = function()
print(g1)  # Out: <generator object function at 0x1012e1888>
```

Notez que le corps d'un générateur n'est **pas** exécuté immédiatement: lorsque vous appelez `function()` dans l'exemple ci-dessus, il retourne immédiatement un objet générateur, sans même exécuter la première instruction d'impression. Cela permet aux générateurs de consommer moins de mémoire que les fonctions qui renvoient une liste et permet de créer des générateurs produisant des séquences infiniment longues.

Pour cette raison, les générateurs sont souvent utilisés dans la science des données et dans d'autres contextes impliquant de grandes quantités de données. Un autre avantage est que l'autre code peut immédiatement utiliser les valeurs fournies par un générateur, sans attendre que la séquence complète soit produite.

Cependant, si vous devez utiliser les valeurs produites par un générateur plusieurs fois et si leur génération coûte plus cher que le stockage, il peut être préférable de stocker les valeurs fournies sous forme de `list` plutôt que de générer à nouveau la séquence. Voir 'Réinitialiser un générateur' ci-dessous pour plus de détails.

Un objet générateur est généralement utilisé dans une boucle ou dans toute fonction nécessitant une itération:

```
for x in g1:
    print("Received", x)

# Output:
```

```
# Received 0
# Received 1
# Received 4
# Received 9
# Received 16
# Received 25
# Received 36
# Received 49
# Received 64
# Received 81

arr1 = list(g1)
# arr1 = [], because the loop above already consumed all the values.
g2 = function()
arr2 = list(g2)  # arr2 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Les objets générateurs étant des itérateurs, on peut les parcourir manuellement en utilisant la fonction `next()` . Cela renverra les valeurs obtenues une par une à chaque invocation suivante.

Sous le capot, chaque fois que vous appelez `next()` sur un générateur, Python exécute des instructions dans le corps de la fonction du générateur jusqu'à ce qu'il atteigne la déclaration de `yield` suivante. À ce stade, il retourne l'argument de la commande de `yield` et se souvient du point où cela s'est produit. L'appel `next()` reprendra l'exécution à partir de ce point et continuera jusqu'à la déclaration de `yield` suivante.

Si Python atteint la fin de la fonction du générateur sans plus de `yield` s, une exception `StopIteration` est `StopIteration` (ceci est normal, tous les itérateurs se comportent de la même manière).

```
g3 = function()
a = next(g3)  # a becomes 0
b = next(g3)  # b becomes 1
c = next(g3)  # c becomes 2
...
j = next(g3)  # Raises StopIteration, j remains undefined
```

Notez que dans Python 2, les objets du générateur avaient des méthodes `.next()` qui pouvaient être utilisées pour parcourir manuellement les valeurs générées. Dans Python 3, cette méthode a été remplacée par la norme `.__next__()` pour tous les itérateurs.

Réinitialisation d'un générateur

Rappelez-vous que vous ne pouvez parcourir que les objets générés par un générateur *une fois* . Si vous avez déjà parcouru les objets dans un script, toute nouvelle tentative à cet effet aboutira à `None` .

Si vous devez utiliser plusieurs fois les objets générés par un générateur, vous pouvez soit définir à nouveau la fonction du générateur et l'utiliser une seconde fois, ou vous pouvez également stocker la sortie de la fonction du générateur dans une liste lors de la première utilisation. La redéfinition de la fonction du générateur sera une bonne option si vous traitez de gros volumes de données et que le stockage d'une liste de tous les éléments de données prendrait beaucoup d'espace disque. À l'inverse, s'il est coûteux de générer les éléments initialement, vous pouvez

préférer les stocker dans une liste pour pouvoir les réutiliser.

Utiliser un générateur pour trouver les numéros de Fibonacci

Un cas d'utilisation pratique d'un générateur consiste à parcourir les valeurs d'une série infinie. Voici un exemple de recherche des dix premiers termes de la séquence de Fibonacci .

```
def fib(a=0, b=1):
    """Generator that yields Fibonacci numbers. `a` and `b` are the seed values"""
    while True:
        yield a
        a, b = b, a + b

f = fib()
print(', '.join(str(next(f)) for _ in range(10)))
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Séquences infinies

Les générateurs peuvent être utilisés pour représenter des séquences infinies:

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1

natural_numbers = integers_starting_from(1)
```

Une séquence infinie de nombres comme ci-dessus peut également être générée à l'aide de `itertools.count` . Le code ci-dessus pourrait être écrit comme ci-dessous

```
natural_numbers = itertools.count(1)
```

Vous pouvez utiliser les compréhensions de générateur sur des générateurs infinis pour produire de nouveaux générateurs:

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

Sachez qu'un générateur infini n'a pas de fin, donc le transmettre à une fonction qui essaiera de consommer le générateur entièrement aura **des conséquences désastreuses** :

```
list(multiples_of_two)  # will never terminate, or raise an OS-specific error
```

Au lieu de cela, utilisez list / set compréhensions avec `range` (ou `xrange` pour python <3.0):

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

ou utilisez `itertools.islice()` pour découper l'itérateur en un sous-ensemble:

```
from itertools import islice
multiples_of_four = (x * 4 for x in integers_starting_from(1))
first_five_multiples_of_four = list(islice(multiples_of_four, 5))
# [4, 8, 12, 16, 20]
```

Notez que le générateur d'origine est également mis à jour, comme tous les autres générateurs provenant de la même racine:

```
next(natural_numbers)    # yields 16
next(multiples_of_two)   # yields 34
next(multiples_of_four)  # yields 24
```

Une séquence infinie peut également être itérée avec un `for` -loop . Assurez-vous d'inclure une instruction de `break` conditionnelle pour que la boucle se termine éventuellement:

```
for idx, number in enumerate(multiplies_of_two):
    print(number)
    if idx == 9:
        break  # stop after taking the first 10 multiplies of two
```

Exemple classique - Numéros de Fibonacci

```
import itertools

def fibonacci():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b

first_ten_fibs = list(itertools.islice(fibonacci(), 10))
# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

def nth_fib(n):
    return next(itertools.islice(fibonacci(), n - 1, n))

ninety_nineth_fib = nth_fib(99)  # 354224848179261915075
```

Céder toutes les valeurs d'une autre itération

Python 3.x 3.3

Utilisez le `yield from` si vous souhaitez générer toutes les valeurs d'une autre itération:

```
def foob(x):
    yield from range(x * 2)
    yield from range(2)

list(foob(5))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

Cela fonctionne également avec des générateurs.

```
def fibto(n):
    a, b = 1, 1
    while True:
        if a >= n: break
        yield a
        a, b = b, a + b

def usefib():
    yield from fibto(10)
    yield from fibto(20)

list(usefib())  # [1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8, 13]
```

Coroutines

Les générateurs peuvent être utilisés pour implémenter des coroutines:

```
# create and advance generator to the first yield
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return start

# example coroutine
@coroutine
def adder(sum = 0):
    while True:
        x = yield sum
        sum += x

# example use
s = adder()
s.send(1) # 1
s.send(2) # 3
```

Les coroutines sont couramment utilisées pour implémenter des machines à états, car elles sont principalement utiles pour créer des procédures à méthode unique nécessitant un état pour fonctionner correctement. Ils opèrent sur un état existant et renvoient la valeur obtenue à l'issue de l'opération.

Rendement avec récursivité: liste récursive de tous les fichiers d'un répertoire

Tout d'abord, importez les bibliothèques qui fonctionnent avec les fichiers:

```
from os import listdir
from os.path import isfile, join, exists
```

Une fonction d'assistance pour lire uniquement les fichiers d'un répertoire:

```
def get_files(path):
    for file in listdir(path):
        full_path = join(path, file)
        if isfile(full_path):
            if exists(full_path):
                yield full_path
```

Une autre fonction d'assistance pour obtenir uniquement les sous-répertoires:

```
def get_directories(path):
    for directory in listdir(path):
        full_path = join(path, directory)
        if not isfile(full_path):
            if exists(full_path):
                yield full_path
```

Maintenant, utilisez ces fonctions pour récupérer tous les fichiers dans un répertoire et tous ses sous-répertoires (en utilisant des générateurs):

```
def get_files_recursive(directory):
    for file in get_files(directory):
        yield file
    for subdirectory in get_directories(directory):
        for file in get_files_recursive(subdirectory): # here the recursive call
            yield file
```

Cette fonction peut être simplifiée en utilisant le `yield from` :

```
def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)
```

Itérer sur les générateurs en parallèle

Pour itérer plusieurs générateurs en parallèle, utilisez le `zip` intégré:

```
for x, y in zip(a,b):
    print(x,y)
```

Résulte en:

```
1 x
2 y
3 z
```

En python 2, utilisez plutôt `itertools.izip` . Ici, nous pouvons également voir que toutes les fonctions `zip` donnent des tuples.

Notez que zip cessera d'itérer dès que l'une des itérables sera à court d'éléments. Si vous souhaitez effectuer une itération aussi longue que la plus longue itération, utilisez

`itertools.zip_longest()` .

Code de construction de refactoring

Supposons que vous ayez un code complexe qui crée et renvoie une liste en commençant par une liste vide et en y ajoutant de manière répétée:

```
def create():
    result = []
    # logic here...
    result.append(value) # possibly in several places
    # more logic...
    return result # possibly in several places

values = create()
```

Lorsqu'il n'est pas pratique de remplacer la logique interne par une compréhension de liste, vous pouvez transformer l'intégralité de la fonction en générateur, puis collecter les résultats:

```
def create_gen():
    # logic...
    yield value
    # more logic
    return # not needed if at the end of the function, of course

values = list(create_gen())
```

Si la logique est récursive, utilisez le `yield from` pour inclure toutes les valeurs de l'appel récursif dans un résultat "aplati":

```
def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)
```

Recherche

La fonction `next` est utile même sans itération. Passer une expression de générateur à `next` est un moyen rapide de rechercher la première occurrence d'un élément correspondant à un prédicat. Code procédural comme

```
def find_and_transform(sequence, predicate, func):
    for element in sequence:
        if predicate(element):
            return func(element)
    raise ValueError

item = find_and_transform(my_sequence, my_predicate, my_func)
```

peut être remplacé par:

```
item = next(my_func(x) for x in my_sequence if my_predicate(x))
```

```
# StopIteration will be raised if there are no matches; this exception can
# be caught and transformed, if desired.
```

À cette fin, il peut être souhaitable de créer un alias, tel que `first = next` , ou une fonction wrapper pour convertir l'exception:

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

Lire Générateurs en ligne: https://riptutorial.com/fr/python/topic/292/generateurs

Chapitre 83: Gestionnaires de contexte (déclaration «avec»)

Introduction

Bien que les gestionnaires de contexte de Python soient largement utilisés, peu de personnes comprennent le but de leur utilisation. Ces instructions, couramment utilisées avec les fichiers de lecture et d'écriture, aident l'application à conserver la mémoire système et à améliorer la gestion des ressources en garantissant que certaines ressources ne sont utilisées que pour certains processus. Cette rubrique explique et illustre l'utilisation des gestionnaires de contexte de Python.

Syntaxe

- avec "context_manager" (comme "alias") (, "context_manager" (comme "alias")?) *:

Remarques

Les gestionnaires de contexte sont définis dans PEP 343 . Ils sont destinés à être utilisés comme mécanisme plus succinct pour la gestion des ressources que d' `try ... finally` construire. La définition formelle est la suivante.

> Dans ce PEP, les gestionnaires de contexte fournissent les `__enter__()` et `__exit__()` qui sont appelées à l'entrée et à la sortie du corps de l'instruction with.

Il continue ensuite à définir l'instruction `with` comme suit.

```
with EXPR as VAR:
    BLOCK
```

La traduction de la déclaration ci-dessus est la suivante:

```
   mgr = (EXPR)
   exit = type(mgr).__exit__  # Not calling it yet
   value = type(mgr).__enter__(mgr)
   exc = True
   try:
       try:
           VAR = value  # Only if "as VAR" is present
           BLOCK
       except:
           # The exceptional case is handled here
           exc = False
           if not exit(mgr, *sys.exc_info()):
               raise
           # The exception is swallowed if exit() returns true
   finally:
       # The normal and non-local-goto cases are handled here
```

```
        if exc:
            exit(mgr, None, None, None)
```

Examples

Introduction aux gestionnaires de contexte et à l'énoncé with

Un gestionnaire de contexte est un objet qui est notifié lorsqu'un contexte (un bloc de code) *commence* et *se termine* . Vous en utilisez généralement un avec l'instruction `with` . Il prend soin de la notification.

Par exemple, les objets de fichier sont des gestionnaires de contexte. Lorsqu'un contexte se termine, l'objet fichier est automatiquement fermé:

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

L'exemple ci-dessus est généralement simplifié en utilisant le mot-clé `as` :

```
with open(filename) as open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Tout ce qui met fin à l'exécution du bloc provoque l'appel de la méthode de sortie du gestionnaire de contexte. Cela inclut des exceptions et peut être utile lorsqu'une erreur vous oblige à quitter prématurément un fichier ouvert ou une connexion. Quitter un script sans fermer correctement les fichiers / connexions est une mauvaise idée, susceptible de provoquer une perte de données ou d'autres problèmes. En utilisant un gestionnaire de contexte, vous pouvez vous assurer que des précautions sont toujours prises pour éviter les dommages ou les pertes de cette manière. Cette fonctionnalité a été ajoutée dans Python 2.5.

Affectation à une cible

De nombreux gestionnaires de contexte renvoient un objet lorsqu'ils sont entrés. Vous pouvez affecter cet objet à un nouveau nom dans l'instruction `with` .

Par exemple, l'utilisation d'une connexion de base de données dans une instruction `with` pourrait vous donner un objet curseur:

```
with database_connection as cursor:
    cursor.execute(sql_query)
```

Les objets fichiers retournent eux-mêmes, cela permet à la fois d'ouvrir l'objet fichier et de l'utiliser comme gestionnaire de contexte dans une expression:

```
with open(filename) as open_file:
    file_contents = open_file.read()
```

Ecrire votre propre gestionnaire de contexte

Un gestionnaire de contexte est un objet qui implémente deux méthodes magiques `__enter__()` et `__exit__()` (bien qu'il puisse également implémenter d'autres méthodes):

```
class AContextManager():

    def __enter__(self):
        print("Entered")
        # optionally return an object
        return "A-instance"

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exited" + (" (with an exception)" if exc_type else ""))
        # return True if you want to suppress the exception
```

Si le contexte sort avec une exception, les informations sur cette exception sera adoptée en tant que triple `exc_type` , `exc_value` , `traceback` (ce sont les mêmes variables que retournée par la `sys.exc_info()` fonction). Si le contexte se ferme normalement, ces trois arguments seront `None` .

Si une exception se produit et est transmise à la méthode `__exit__` , la méthode peut renvoyer `True` afin de supprimer l'exception ou l'exception sera relancée à la fin de la fonction `__exit__` .

```
with AContextManager() as a:
    print("a is %r" % a)
# Entered
# a is 'A-instance'
# Exited

with AContextManager() as a:
    print("a is %d" % a)
# Entered
# Exited (with an exception)
# Traceback (most recent call last):
#   File "<stdin>", line 2, in <module>
# TypeError: %d format: a number is required, not str
```

Notez que dans le deuxième exemple, même si une exception se produit au milieu du corps de l'instruction with-statement, le gestionnaire `__exit__` toujours exécuté avant que l'exception ne se propage à la portée externe.

Si vous n'avez besoin que d'une méthode `__exit__` , vous pouvez renvoyer l'instance du gestionnaire de contexte:

```
class MyContextManager:
    def __enter__(self):
        return self

    def __exit__(self):
        print('something')
```

Ecrire votre propre gestionnaire de contexte en utilisant la syntaxe du générateur

Il est également possible d'écrire un gestionnaire de contexte en utilisant la syntaxe du générateur grâce au décorateur `contextlib.contextmanager` :

```
import contextlib

@contextlib.contextmanager
def context_manager(num):
    print('Enter')
    yield num + 1
    print('Exit')

with context_manager(2) as cm:
    # the following instructions are run when the 'yield' point of the context
    # manager is reached.
    # 'cm' will have the value that was yielded
    print('Right in the middle with cm = {}'.format(cm))
```

produit:

```
Enter
Right in the middle with cm = 3
Exit
```

Le décorateur simplifie l'écriture d'un gestionnaire de contexte en convertissant un générateur en un seul. Avant que l'expression de rendement ne devienne la méthode `__enter__` , la valeur `__enter__` devient la valeur renvoyée par le générateur (qui peut être liée à une variable dans l'instruction with), et tout ce qui suit l'expression de rendement devient la méthode `__exit__` .

Si une exception doit être gérée par le gestionnaire de contexte, un `try..except..finally` -block peut être écrit dans le générateur et toute exception déclenchée dans le bloc `with` -block sera traitée par ce bloc d'exception.

```
@contextlib.contextmanager
def error_handling_context_manager(num):
    print("Enter")
    try:
        yield num + 1
    except ZeroDivisionError:
        print("Caught error")
    finally:
        print("Cleaning up")
    print("Exit")

with error_handling_context_manager(-1) as cm:
    print("Dividing by cm = {}".format(cm))
    print(2 / cm)
```

Cela produit:

```
Enter
```

```
Dividing by cm = 0
Caught error
Cleaning up
Exit
```

Plusieurs gestionnaires de contexte

Vous pouvez ouvrir plusieurs gestionnaires de contenu en même temps:

```
with open(input_path) as input_file, open(output_path, 'w') as output_file:

    # do something with both files.

    # e.g. copy the contents of input_file into output_file
    for line in input_file:
        output_file.write(line + '\n')
```

Il a le même effet que les gestionnaires de contexte d'imbrication:

```
with open(input_path) as input_file:
    with open(output_path, 'w') as output_file:
        for line in input_file:
            output_file.write(line + '\n')
```

Gérer les ressources

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):
        self.open_file.close()
```

`__init__()` configure l'objet, en définissant le nom du fichier et le mode pour ouvrir le fichier. `__enter__()` s'ouvre et renvoie le fichier et `__exit__()` ferme.

L'utilisation de ces méthodes magiques (`__enter__` , `__exit__`) vous permet d'implémenter des objets facilement utilisables `with` l'instruction with.

Utiliser la classe de fichier:

```
for _ in range(10000):
    with File('foo.txt', 'w') as f:
        f.write('foo')
```

Lire Gestionnaires de contexte (déclaration «avec») en ligne:
https://riptutorial.com/fr/python/topic/928/gestionnaires-de-contexte--declaration--avec--

Chapitre 84: hashlib

Introduction

hashlib implémente une interface commune à de nombreux algorithmes de hachage et de résumé de messages sécurisés. Les algorithmes de hachage sécurisés FIPS SHA1, SHA224, SHA256, SHA384 et SHA512 sont inclus.

Examples

MD5 hash d'une chaîne

Ce module implémente une interface commune à de nombreux algorithmes de hachage et de résumé des messages sécurisés. Sont inclus les algorithmes de hachage sécurisés FIPS SHA1, SHA224, SHA256, SHA384 et SHA512 (définis dans FIPS 180-2) ainsi que l'algorithme MD5 de RSA (défini dans Internet RFC 1321).

Il existe une méthode de constructeur nommée pour chaque type de hachage. Tous renvoient un objet de hachage avec la même interface simple. Par exemple, utilisez `sha1()` pour créer un objet de hachage SHA1.

```
hash.sha1()
```

Les constructeurs des algorithmes de hachage qui sont toujours présents dans ce module sont `md5()` , `sha1()` , `sha224()` , `sha256()` , `sha384()` et `sha512()` .

Vous pouvez maintenant alimenter cet objet avec des chaînes arbitraires en utilisant la méthode `update()` . A tout moment, vous pouvez demander le résumé de la concaténation des chaînes qui lui ont été `hexdigest()` à l'aide des méthodes `digest()` ou `hexdigest()` .

```
hash.update(arg)
```

> Mettez à jour l'objet hash avec la chaîne arg. Les appels répétés équivalent à un seul appel avec la concaténation de tous les arguments: m.update (a); m.update (b) est équivalent à m.update (a + b).

```
hash.digest()
```

> Renvoie le résumé des chaînes passées à la méthode update () jusqu'à présent. Il s'agit d'une chaîne d'octets digest_size pouvant contenir des caractères non-ASCII, y compris des octets nuls.

```
hash.hexdigest()
```

> Comme digest (), sauf que digest est renvoyé sous la forme d'une chaîne de longueur

double, contenant uniquement des chiffres hexadécimaux. Cela peut être utilisé pour échanger la valeur en toute sécurité dans le courrier électronique ou dans d'autres environnements non binaires.

Voici un exemple:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.hexdigest()
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
>>> m.digest_size
16
>>> m.block_size
64
```

ou:

```
hashlib.md5("Nobody inspects the spammish repetition").hexdigest()
    'bb649c83dd1ea5c9d9dec9a18df0ffe9'
```

algorithme fourni par OpenSSL

Un constructeur `new()` générique qui prend le nom de chaîne de l'algorithme souhaité comme premier paramètre existe également pour permettre l'accès aux hachages répertoriés ci-dessus, ainsi que tout autre algorithme que votre bibliothèque OpenSSL peut offrir. Les constructeurs nommés sont beaucoup plus rapides que `new()` et devraient être préférés.

Utiliser `new()` avec un algorithme fourni par OpenSSL:

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Lire hashlib en ligne: https://riptutorial.com/fr/python/topic/8980/hashlib

Chapitre 85: ijson

Introduction

ijson est une excellente bibliothèque pour travailler avec des fichiers JSON en Python. Malheureusement, par défaut, il utilise un analyseur JSON Python pur comme backend. Des performances beaucoup plus élevées peuvent être obtenues en utilisant un backend C.

Examples

Exemple simple

Exemple Exemple Tiré d'une analyse comparative

```
import ijson

def load_json(filename):
    with open(filename, 'r') as fd:
        parser = ijson.parse(fd)
        ret = {'builders': {}}
        for prefix, event, value in parser:
            if (prefix, event) == ('builders', 'map_key'):
                buildername = value
                ret['builders'][buildername] = {}
            elif prefix.endswith('.shortname'):
                ret['builders'][buildername]['shortname'] = value

        return ret

if __name__ == "__main__":
    load_json('allthethings.json')
```

JSON FILE LINK

Lire ijson en ligne: https://riptutorial.com/fr/python/topic/8342/ijson

Chapitre 86: Implémentations non officielles de Python

Examples

IronPython

Implémentation open source pour .NET et Mono écrite en C #, sous licence Apache 2.0. Il s'appuie sur DLR (Dynamic Language Runtime). Il ne supporte que la version 2.7, la version 3 est en cours de développement.

Différences avec CPython:

- Intégration étroite avec .NET Framework.
- Les chaînes sont Unicode par défaut.
- Ne prend pas en charge les extensions pour CPython écrites en C.
- Ne souffre pas de Global Interpreter Lock.
- Les performances sont généralement inférieures, même si cela dépend des tests.

Bonjour le monde

```
print "Hello World!"
```

Vous pouvez également utiliser les fonctions .NET:

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

Liens externes

- Site officiel
- Dépôt GitHub

Jython

Implémentation open source pour JVM écrite en Java, sous licence Python Software Foundation. Il ne supporte que la version 2.7, la version 3 est en cours de développement.

Différences avec CPython:

- Intégration étroite avec JVM.

- Les chaînes sont en Unicode.
- Ne prend pas en charge les extensions pour CPython écrites en C.
- Ne souffre pas de Global Interpreter Lock.
- Les performances sont généralement inférieures, même si cela dépend des tests.

Bonjour le monde

```
print "Hello World!"
```

Vous pouvez également utiliser les fonctions Java:

```
from java.lang import System
System.out.println("Hello World!")
```

Liens externes

- Site officiel
- Dépôt mercuriel

Transcrypt

Transcrypt est un outil permettant de précompiler un sous-ensemble assez étendu de Python en Javascript compact et lisible. Il présente les caractéristiques suivantes:

- Permet une programmation OO classique avec héritage multiple en utilisant la syntaxe Python pure, analysée par l'analyseur natif de CPython
- Intégration transparente avec l'univers des bibliothèques JavaScript orientées Web de haute qualité, plutôt qu'avec les bibliothèques Python orientées bureau
- Système de module hiérarchique basé sur URL permettant la distribution de modules via PyPi
- Relation simple entre le source Python et le code JavaScript généré pour un débogage facile
- Mappages multiniveaux et annotation facultative du code cible avec les références source
- Téléchargements compacts, Ko plutôt que MB
- Code JavaScript optimisé, utilisant la mémorisation (mise en cache des appels) pour éventuellement contourner la chaîne de recherche du prototype
- La surcharge de l'opérateur peut être activée ou désactivée localement pour faciliter la lecture mathématique lisible

Code taille et vitesse

L'expérience a montré que 650 ko de code source Python se traduit approximativement par la même quantité de code source JavaScript. La vitesse correspond à la vitesse de JavaScript manuscrit et peut la dépasser si la mémorisation des appels est activée.

Intégration avec HTML

```
<script src="__javascript__/hello.js"></script>
<h2>Hello demo</h2>

<p>
<div id = "greet">...</div>
<button onclick="hello.solarSystem.greet ()">Click me repeatedly!</button>

<p>
<div id = "explain">...</div>
<button onclick="hello.solarSystem.explain ()">And click me repeatedly too!</button>
```

Intégration avec JavaScript et DOM

```
from itertools import chain

class SolarSystem:
    planets = [list (chain (planet, (index + 1,))) for index, planet in enumerate ((
        ('Mercury', 'hot', 2240),
        ('Venus', 'sulphurous', 6052),
        ('Earth', 'fertile', 6378),
        ('Mars', 'reddish', 3397),
        ('Jupiter', 'stormy', 71492),
        ('Saturn', 'ringed', 60268),
        ('Uranus', 'cold', 25559),
        ('Neptune', 'very cold', 24766)
    ))]

    lines = (
        '{} is a {} planet',
        'The radius of {} is {} km',
        '{} is planet nr. {} counting from the sun'
    )

    def __init__ (self):
        self.lineIndex = 0

    def greet (self):
        self.planet = self.planets [int (Math.random () * len (self.planets))]
        document.getElementById ('greet') .innerHTML = 'Hello {}'.format (self.planet [0])
        self.explain ()

    def explain (self):
        document.getElementById ('explain').innerHTML = (
            self.lines [self.lineIndex] .format (self.planet [0], self.planet [self.lineIndex
+ 1])
        )
        self.lineIndex = (self.lineIndex + 1) % 3
         solarSystem = SolarSystem ()
```

Intégration avec d'autres bibliothèques

JavaScript

Transcrypt peut être utilisé en combinaison avec toute bibliothèque JavaScript sans mesures ni syntaxe spéciales. Dans la documentation, des exemples sont donnés pour ao react.js, riot.js, fabric.js et node.js.

Relation entre Python et le code JavaScript

Python

```
class A:
    def __init__ (self, x):
        self.x = x

    def show (self, label):
        print ('A.show', label, self.x)

class B:
    def __init__ (self, y):
        alert ('In B constructor')
        self.y = y

    def show (self, label):
        print ('B.show', label, self.y)

class C (A, B):
    def __init__ (self, x, y):
        alert ('In C constructor')
        A.__init__ (self, x)
        B.__init__ (self, y)
        self.show ('constructor')

    def show (self, label):
        B.show (self, label)
        print ('C.show', label, self.x, self.y)

a = A (1001)
a.show ('america')

b = B (2002)
b.show ('russia')

c = C (3003, 4004)
c.show ('netherlands')

show2 = c.show
show2 ('copy')
```

JavaScript

```
var A = __class__ ('A', [object], {
    get __init__ () {return __get__ (this, function (self, x) {
        self.x = x;
    });},
```

```
    get show () {return __get__ (this, function (self, label) {
        print ('A.show', label, self.x);
    });}
});
var B = __class__ ('B', [object], {
    get __init__ () {return __get__ (this, function (self, y) {
        alert ('In B constructor');
        self.y = y;
    });},
    get show () {return __get__ (this, function (self, label) {
        print ('B.show', label, self.y);
    });}
});
var C = __class__ ('C', [A, B], {
    get __init__ () {return __get__ (this, function (self, x, y) {
        alert ('In C constructor');
        A.__init__ (self, x);
        B.__init__ (self, y);
        self.show ('constructor');
    });},
    get show () {return __get__ (this, function (self, label) {
        B.show (self, label);
        print ('C.show', label, self.x, self.y);
    });}
});
var a = A (1001);
a.show ('america');
var b = B (2002);
b.show ('russia');
var c = C (3003, 4004);
c.show ('netherlands');
var show2 = c.show;
show2 ('copy');
```

Liens externes

- Site officiel: http://www.transcrypt.org/
- Dépôt: https://github.com/JdeH/Transcrypt

Lire Implémentations non officielles de Python en ligne:
https://riptutorial.com/fr/python/topic/5225/implementations-non-officielles-de-python

Chapitre 87: Importation de modules

Syntaxe

- import *nom_module*
- import *nom_module.submodule_name*
- à partir de *nom_module* import *
- de *submodule_name* d'importation *module_name* [, *class_name, nom_fonction,* etc ...]
- from *nom_module* import *some_name* as *new_name*
- de *nom_classe* importation *module_name.submodule_name* [, *nom_fonction,* etc ...]

Remarques

L'importation d'un module fera que Python évalue tous les codes de niveau supérieur de ce module afin qu'il *apprenne* toutes les fonctions, les classes et les variables qu'il contient. Lorsque vous voulez importer un module ailleurs, faites attention à votre code de premier niveau et encapsulez-le dans `if __name__ == '__main__':` si vous ne voulez pas qu'il soit exécuté lors de l'importation du module.

Examples

Importer un module

Utilisez l'instruction `import` :

```
>>> import random
>>> print(random.randint(1, 10))
4
```

`import module` importation importera un module puis vous permettra de référencer ses objets - valeurs, fonctions et classes, par exemple - en utilisant la syntaxe `module.name` . Dans l'exemple ci-dessus, le module `random` est importé, qui contient la fonction `randint` . Donc, en important `random` `randint` `random` vous pouvez appeler `randint` avec `random.randint` .

Vous pouvez importer un module et lui attribuer un nom différent:

```
>>> import random as rn
>>> print(rn.randint(1, 10))
4
```

Si votre fichier python `main.py` est dans le même dossier que `custom.py` . Vous pouvez l'importer comme ceci:

```
import custom
```

Il est également possible d'importer une fonction depuis un module:

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

Pour importer des fonctions spécifiques plus profondément dans un module, l'opérateur point peut être utilisé **uniquement** du côté gauche du mot clé `import` :

```
from urllib.request import urlopen
```

En python, nous avons deux façons d'appeler la fonction du niveau supérieur. L'un est `import` et l'autre `from` . Nous devrions utiliser l' `import` lorsque nous avons une possibilité de collision de noms. Supposons que nous `hello.py` fichiers `hello.py` et `world.py` ayant la même fonction nommée `function` . Ensuite, `import` déclaration d' `import` fonctionnera bien.

```
from hello import function
from world import function

function() #world's function will be invoked. Not hello's
```

En général, l' `import` vous fournira un espace de noms.

```
import hello
import world

hello.function() # exclusively hello's function will be invoked
world.function() # exclusively world's function will be invoked
```

Mais si vous êtes bien sûr, dans votre projet tout il n'y a aucun moyen ayant même nom de fonction , vous devez utiliser `from` déclaration

Plusieurs importations peuvent être effectuées sur la même ligne:

```
>>> # Multiple modules
>>> import time, sockets, random
>>> # Multiple functions
>>> from math import sin, cos, tan
>>> # Multiple constants
>>> from math import pi, e

>>> print(pi)
3.141592653589793
>>> print(cos(45))
0.5253219888177297
>>> print(time.time())
1482807222.7240417
```

Les mots-clés et la syntaxe présentés ci-dessus peuvent également être utilisés dans des combinaisons:

```
>>> from urllib.request import urlopen as geturl, pathname2url as path2url, getproxies
```

```
>>> from math import factorial as fact, gamma, atan as arctan
>>> import random.randint, time, sys

>>> print(time.time())
1482807222.7240417
>>> print(arctan(60))
1.554131203080956
>>> filepath = "/dogs/jumping poodle (december).png"
>>> print(path2url(filepath))
/dogs/jumping%20poodle%20%28december%29.png
```

Importation de noms spécifiques à partir d'un module

Au lieu d'importer le module complet, vous ne pouvez importer que les noms spécifiés:

```
from random import randint # Syntax "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))      # Out: 5
```

`from random` est nécessaire, car l'interpréteur python doit savoir de quelle ressource il doit importer une fonction ou une classe et `import randint` spécifie la fonction ou la classe elle-même.

Un autre exemple ci-dessous (similaire à celui ci-dessus):

```
from math import pi
print(pi)                  # Out: 3.14159265359
```

L'exemple suivant génère une erreur, car nous n'avons pas importé de module:

```
random.randrange(1, 10)    # works only if "import random" has been run before
```

Les sorties:

```
NameError: name 'random' is not defined
```

L'interpréteur Python ne comprend pas ce que vous entendez par `random` . Il doit être déclaré en ajoutant l' `import random` à l'exemple:

```
import random
random.randrange(1, 10)
```

Importer tous les noms d'un module

```
from module_name import *
```

par exemple:

```
from math import *
sqrt(2)    # instead of math.sqrt(2)
ceil(2.7)  # instead of math.ceil(2.7)
```

Cela importera tous les noms définis dans le module `math` dans l'espace de noms global, à l'exception des noms commençant par un trait de soulignement (ce qui indique que le rédacteur estime qu'il est destiné à un usage interne uniquement).

Avertissement : Si une fonction du même nom a déjà été définie ou importée, elle sera **écrasée** . Importer presque toujours uniquement des noms spécifiques à `from math import sqrt, ceil` est la **méthode recommandée** :

```
def sqrt(num):
    print("I don't know what's the square root of {}.".format(num))

sqrt(4)
# Output: I don't know what's the square root of 4.

from math import *
sqrt(4)
# Output: 2.0
```

Les importations étoilées ne sont autorisées qu'au niveau du module. Les tentatives de les exécuter dans des définitions de classe ou de fonction entraînent une `SyntaxError` .

```
def f():
    from math import *
```

et

```
class A:
    from math import *
```

les deux échouent avec:

```
SyntaxError: import * only allowed at module level
```

La variable spéciale __all__

Les modules peuvent avoir une variable spéciale nommée `__all__` pour restreindre les variables importées lors de l'utilisation `from mymodule import *` .

Vu le module suivant:

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

Seul `imported_by_star` est importé lors de l'utilisation `from mymodule import *` :

```
>>> from mymodule import *
```

```
>>> imported_by_star
42
>>> not_imported_by_star
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'not_imported_by_star' is not defined
```

Cependant, `not_imported_by_star` peut être importé explicitement:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
21
```

Importation programmatique

Python 2.x 2.7

Pour importer un module via un appel de fonction, utilisez le module `importlib` (inclus dans Python à partir de la version 2.7):

```
import importlib
random = importlib.import_module("random")
```

La fonction `importlib.import_module()` importera également le sous-module d'un paquet directement:

```
collections_abc = importlib.import_module("collections.abc")
```

Pour les anciennes versions de Python, utilisez le module `imp` .

Python 2.x 2.7

Utilisez les fonctions `imp.find_module` et `imp.load_module` pour effectuer une importation par programme.

Tiré de la documentation standard de la bibliothèque

```
import imp, sys
def import_module(name):
    fp, pathname, description = imp.find_module(name)
    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        if fp:
            fp.close()
```

N'utilisez **PAS** `__import__()` pour importer des modules par programmation! Il y a des détails subtils concernant `sys.modules` , l'argument `fromlist` , etc., qui sont faciles à ignorer et que `importlib.import_module()` gère pour vous.

Importer des modules à partir d'un emplacement de système de fichiers

arbitraire

Si vous souhaitez importer un module qui n'existe pas déjà en tant que module intégré dans la bibliothèque standard de Python ni en tant que package, vous pouvez le faire en ajoutant le chemin d'accès au répertoire dans lequel votre module est trouvé dans `sys.path` . Cela peut être utile lorsque plusieurs environnements Python existent sur un hôte.

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

Il est important que vous ajoutiez le chemin d'accès au *répertoire* dans lequel `mymodule` est trouvé, pas le chemin du module lui-même.

Règles PEP8 pour les importations

Quelques recommandations de style PEP8 pour les importations:

1. Les importations doivent être sur des lignes séparées:

   ```
   from math import sqrt, ceil      # Not recommended
   from math import sqrt            # Recommended
   from math import ceil
   ```

2. Ordre des importations comme suit en haut du module:

   - Importation de bibliothèque standard
   - Importations de tiers liées
   - Importations spécifiques aux applications / bibliothèques locales

3. Les importations de caractères génériques doivent être évitées car elles entraînent une confusion dans les noms de l'espace de noms actuel. Si vous effectuez une `from module import *` , il peut être difficile de savoir si un nom spécifique dans votre code provient du `module` ou non. Ceci est doublement vrai si vous avez plusieurs instructions de type `from module import *` .

4. Évitez d'utiliser des importations relatives; utiliser des importations explicites à la place.

Importer des sous-modules

```
from module.submodule import function
```

Cette `function` importation de `module.submodule` .

__import __ () fonction

La fonction `__import__()` peut être utilisée pour importer des modules dont le nom n'est connu qu'à l'exécution

```
if user_input == "os":
    os = __import__("os")

# equivalent to import os
```

Cette fonction peut également être utilisée pour spécifier le chemin du fichier vers un module

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

Réimporter un module

Lors de l'utilisation de l'interpréteur interactif, vous souhaiterez peut-être recharger un module. Cela peut être utile si vous modifiez un module et que vous souhaitez importer la version la plus récente, ou si vous avez appliqué un patch sur un élément d'un module existant et que vous souhaitez annuler vos modifications.

Notez que vous **ne pouvez pas** `import` le module à nouveau pour annuler:

```
import math
math.pi = 3
print(math.pi)    # 3
import math
print(math.pi)    # 3
```

C'est parce que l'interpréteur enregistre chaque module que vous importez. Et lorsque vous essayez de réimporter un module, l'interprète le voit dans le registre et ne fait rien. Donc, le moyen difficile de réimporter est d'utiliser l' `import` après avoir supprimé l'élément correspondant du registre:

```
print(math.pi)    # 3
import sys
if 'math' in sys.modules:  # Is the ``math`` module in the register?
    del sys.modules['math']  # If so, remove it.
import math
print(math.pi)    # 3.141592653589793
```

Mais il y a plus d'une manière simple et directe.

Python 2

Utilisez la fonction de `reload` :

Python 2.x 2.3

```
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

Python 3

La fonction de `reload` a été déplacée vers `importlib` :

Python 3.x 3.0

```
import math
math.pi = 3
print(math.pi)    # 3
from importlib import reload
reload(math)
print(math.pi)    # 3.141592653589793
```

Lire Importation de modules en ligne: https://riptutorial.com/fr/python/topic/249/importation-de-modules

Chapitre 88: Incompatibilités entre Python 2 et Python 3

Introduction

Contrairement à la plupart des langages, Python prend en charge deux versions principales. Depuis 2008, quand Python 3 est sorti, beaucoup ont fait la transition, alors que beaucoup ne l'ont pas fait. Afin de comprendre les deux, cette section couvre les différences importantes entre Python 2 et Python 3.

Remarques

Il existe actuellement deux versions prises en charge de Python: 2.7 (Python 2) et 3.6 (Python 3). De plus, les versions 3.3 et 3.4 reçoivent les mises à jour de sécurité au format source.

Python 2.7 est rétrocompatible avec la plupart des versions antérieures de Python et peut exécuter le code Python de la plupart des versions 1.x et 2.x de Python sans modification. Il est largement disponible, avec une vaste collection de paquets. Il est également considéré comme obsolète par les développeurs de CPython et ne reçoit que le développement de la sécurité et des correctifs. Les développeurs de CPython ont l'intention d'abandonner cette version du langage en 2020 .

Selon Python Enhancement Proposal 373, il n'y aura pas de versions futures planifiées de Python 2 après le 25 juin 2016, mais les corrections de bogues et les mises à jour de sécurité seront prises en charge jusqu'en 2020 (il ne spécifie pas la date exacte de 2020). 2.)

Python 3 a intentionnellement rompu la compatibilité avec les versions antérieures pour répondre aux préoccupations des développeurs de langage concernant le noyau du langage. Python 3 reçoit de nouveaux développements et de nouvelles fonctionnalités. C'est la version du langage que les développeurs de langue ont l'intention de faire évoluer.

Au fil du temps, entre la version initiale de Python 3.0 et la version actuelle, certaines fonctionnalités de Python 3 ont été transférées dans Python 2.6, et d'autres parties de Python 3 ont été étendues pour avoir une syntaxe compatible avec Python 2. Il est donc possible d'écrire Python qui fonctionnera sur Python 2 et Python 3, en utilisant les futures importations et les modules spéciaux (comme **six**).

Les futures importations doivent être au début de votre module:

```
from __future__ import print_function
# other imports and instructions go after __future__
print('Hello world')
```

Pour plus d'informations sur le module `__future__` , consultez la page correspondante dans la documentation Python .

L' outil 2to3 est un programme Python qui convertit le code Python 2.x en code Python 3.x. Voir aussi la documentation Python .

Le paquet six fournit des utilitaires pour la compatibilité avec Python 2/3:

- accès unifié aux bibliothèques renommées
- variables pour les types chaîne / unicode
- fonctions pour la méthode qui a été supprimée ou a été renommée

Une référence pour les différences entre Python 2 et Python 3 peut être trouvée ici .

Examples

Relevé d'impression ou fonction d'impression

Dans Python 2, `print` est une déclaration:

Python 2.x 2.7

```
print "Hello World"
print                           # print a newline
print "No newline",             # add trailing comma to remove newline
print >>sys.stderr, "Error"     # print to stderr
print("hello")                  # print "hello", since ("hello") == "hello"
print()                         # print an empty tuple "()"
print 1, 2, 3                   # print space-separated arguments: "1 2 3"
print(1, 2, 3)                  # print tuple "(1, 2, 3)"
```

Dans Python 3, `print()` est une fonction, avec des arguments de mots-clés pour des utilisations courantes:

Python 3.x 3.0

```
print "Hello World"              # SyntaxError
print("Hello World")
print()                         # print a newline (must use parentheses)
print("No newline", end="")      # end specifies what to append (defaults to newline)
print("Error", file=sys.stderr)  # file specifies the output buffer
print("Comma", "separated", "output", sep=",")  # sep specifies the separator
print("A", "B", "C", sep="")     # null string for sep: prints as ABC
print("Flush this", flush=True)  # flush the output buffer, added in Python 3.3
print(1, 2, 3)                   # print space-separated arguments: "1 2 3"
print((1, 2, 3))                 # print tuple "(1, 2, 3)"
```

La fonction d'impression a les paramètres suivants:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep` est ce qui sépare les objets que vous passez à imprimer. Par exemple:

```
print('foo', 'bar', sep='~') # out: foo~bar
```

```
print('foo', 'bar', sep='.') # out: foo.bar
```

`end` est la fin de l'instruction print. Par exemple:

```
print('foo', 'bar', end='!') # out: foo bar!
```

Impression à nouveau après une déclaration d'impression de fin de non-retour à la ligne imprimera à la même ligne:

```
print('foo', end='~')
print('bar')
# out: foo~bar
```

Remarque: Pour une compatibilité future, `print` *fonction d'* `print` est également disponible dans Python 2.6 et *supérieur* . Cependant, il ne peut être utilisé que si l'analyse de l' *instruction* `print` est désactivée avec

```
from __future__ import print_function
```

Cette fonction a exactement le même format que celle de Python 3, sauf qu'elle ne contient pas le paramètre `flush` .

Voir PEP 3105 pour la justification.

Chaînes: Octets versus Unicode

Python 2.x 2.7

En Python 2, il existe deux variantes de chaîne: celles faites d'octets de type (`str`) et celles de texte de type (`unicode`).

Dans Python 2, un objet de type `str` est toujours une séquence d'octets, mais est couramment utilisé à la fois pour les données textuelles et binaires.

Un littéral de chaîne est interprété comme une chaîne d'octets.

```
s = 'Cafe'    # type(s) == str
```

Il y a deux exceptions: Vous pouvez définir un *littéral Unicode (text)* explicitement en préfixant le littéral avec `u` :

```
s = u'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == str
```

Vous pouvez également spécifier que les littéraux de chaîne d'un module entier doivent créer des littéraux Unicode (texte):

```
from __future__ import unicode_literals
```

```
s = 'Café'   # type(s) == unicode
b = 'Lorem ipsum'  # type(b) == unicode
```

Pour vérifier si votre variable est une chaîne (Unicode ou une chaîne d'octets), vous pouvez utiliser:

```
isinstance(s, basestring)
```

Python 3.x 3.0

Dans Python 3, le type `str` est un type de texte Unicode.

```
s = 'Cafe'           # type(s) == str
s = 'Café'           # type(s) == str (note the accented trailing e)
```

De plus, Python 3 a ajouté un objet `bytes` , adapté aux "blobs" binaires ou à l'écriture dans des fichiers indépendants du codage. Pour créer un objet octets, vous pouvez ajouter un préfixe `b` à un littéral de chaîne ou appeler la méthode d' `encode` la chaîne:

```
# Or, if you really need a byte string:
s = b'Cafe'          # type(s) == bytes
s = 'Café'.encode()  # type(s) == bytes
```

Pour tester si une valeur est une chaîne, utilisez:

```
isinstance(s, str)
```

Python 3.x 3.3

Il est également possible de préfixer des littéraux de chaîne avec un préfixe `u` pour faciliter la compatibilité entre les bases de code Python 2 et Python 3. Puisque, dans Python 3, toutes les chaînes sont Unicode par défaut, ajouter une chaîne à un littéral avec `u` n'a aucun effet:

```
u'Cafe' == 'Cafe'
```

Le préfixe de chaîne Unicode brut de Python 2, `ur` n'est pas supporté, cependant:

```
>>> ur'Café'
  File "<stdin>", line 1
    ur'Café'
            ^
SyntaxError: invalid syntax
```

Notez que vous devez `encode` un objet text (`str`) Python 3 pour le convertir en une représentation en `bytes` de ce texte. Le codage par défaut de cette méthode est UTF-8 .

Vous pouvez utiliser le `decode` pour demander à un objet `bytes` quel texte Unicode il représente:

```
>>> b.decode()
'Café'
```

Python 2.x 2.6

Bien que le type d' `bytes` existe à la fois dans Python 2 et 3, le type `unicode` n'existe que dans Python 2. Pour utiliser les chaînes Unicode implicites de Python 3 dans Python 2, ajoutez ce qui suit en haut de votre fichier de code:

```
from __future__ import unicode_literals
print(repr("hi"))
# u'hi'
```

Python 3.x 3.0

Une autre différence importante est que l'indexation des octets dans Python 3 donne un résultat `int` comme celui-ci:

```
b"abc"[0] == 97
```

Alors que le découpage en taille de 1 donne un objet de longueur 1 octet:

```
b"abc"[0:1] == b"a"
```

En outre, Python 3 corrige certains comportements inhabituels avec Unicode, à savoir l'inversion des chaînes d'octets dans Python 2. Par exemple, le problème suivant est résolu:

```
# -*- coding: utf8 -*-
print("Hi, my name is Łukasz Langa.")
print(u"Hi, my name is Łukasz Langa."[::-1])
print("Hi, my name is Łukasz Langa."[::-1])

# Output in Python 2
# Hi, my name is Łukasz Langa.
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsaku◆◆ si eman ym ,iH

# Output in Python 3
# Hi, my name is Łukasz Langa.
# .agnaL zsakuŁ si eman ym ,iH
# .agnaL zsakuŁ si eman ym ,iH
```

Division entière

Le **symbole de division** standard (`/`) fonctionne différemment dans Python 3 et Python 2 lorsqu'il est appliqué à des entiers.

Lors de la division d'un entier par un autre entier dans Python 3, l'opération de division `x / y` représente une **division réelle** (utilise la méthode `__truediv__`) et produit un résultat à virgule flottante. Pendant ce temps, la même opération dans Python 2 représente une **division classique** qui arrondit le résultat à l'infini négatif (également appelé prise de *parole*).

Par exemple:

Code	Sortie Python 2	Sortie Python 3
`3 / 2`	1	1,5
`2 / 3`	0	0.6666666666666666
`-3 / 2`	-2	-1.5

Le comportement d'arrondi par rapport à zéro a été déprécié dans Python 2.2 , mais reste dans Python 2.7 pour des raisons de compatibilité ascendante et a été supprimé dans Python 3.

Note: Pour obtenir un résultat *flottant* dans Python 2 (sans arrondi), vous pouvez spécifier l'un des opérandes avec le point décimal. L'exemple ci-dessus de `2/3` qui donne `0` dans Python 2 doit être utilisé comme `2 / 3.0` ou `2.0 / 3` ou `2.0/3.0` pour obtenir `0.6666666666666666`

Code	Sortie Python 2	Sortie Python 3
`3.0 / 2.0`	1,5	1,5
`2 / 3.0`	0.6666666666666666	0.6666666666666666
`-3.0 / 2`	-1.5	-1.5

Il y a aussi l' **opérateur de division par étage** (`//`), qui fonctionne de la même manière dans les deux versions: il est arrondi à l'entier le plus proche. (bien qu'un float soit retourné lorsqu'il est utilisé avec des floats) Dans les deux versions, l'opérateur `//` correspond à `__floordiv__` .

Code	Sortie Python 2	Sortie Python 3
`3 // 2`	1	1
`2 // 3`	0	0
`-3 // 2`	-2	-2
`3.0 // 2.0`	1.0	1.0
`2.0 // 3`	0.0	0.0
`-3 // 2.0`	-2,0	-2,0

On peut explicitement imposer une division réelle ou une division par étage en utilisant des fonctions natives dans le module `operator` :

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25           # equivalent to `/` in Python 3
```

```
assert floordiv(10, 8) == 1             # equivalent to `//`
```

Bien que claire et explicite, l'utilisation des fonctions de l'opérateur pour chaque division peut être fastidieuse. La modification du comportement de l'opérateur `/` sera souvent préférée. Une pratique courante consiste à éliminer le comportement de division typique en ajoutant la `from __future__ import division` comme première instruction de chaque module:

```
# needs to be the first statement in a module
from __future__ import division
```

Code	Sortie Python 2	Sortie Python 3
`3 / 2`	1,5	1,5
`2 / 3`	0.6666666666666666	0.6666666666666666
`-3 / 2`	-1.5	-1.5

`from __future__ import division` garantit que l'opérateur `/` représente la division vraie et uniquement dans les modules qui contiennent l'importation `__future__` , il n'y a donc aucune raison impérieuse de ne pas l'activer dans tous les nouveaux modules.

Remarque : Certains autres langages de programmation *arrondissent vers zéro* (troncature) plutôt que *vers l'infini négatif,* contrairement à Python (dans ces langages `-3 / 2 == -1`). Ce comportement peut créer une confusion lors du portage ou de la comparaison du code.

Note sur les opérandes float : En alternative à la `from __future__ import division` , on pourrait utiliser le symbole de division habituel `/` et s'assurer qu'au moins un des opérandes est un float: `3 / 2.0 == 1.5` . Cependant, cela peut être considéré comme une mauvaise pratique. Il est trop facile d'écrire `average = sum(items) / len(items)` et oublier de lancer l'un des arguments pour flotter. De plus, de tels cas peuvent fréquemment échapper à la connaissance pendant le test, par exemple, si vous testez un tableau contenant des `float` mais recevez un tableau de données `int` en production. De plus, si le même code est utilisé dans Python 3, les programmes qui s'attendent à ce que `3 / 2 == 1` soit vrai ne fonctionneront pas correctement.

Voir PEP 238 pour une explication plus détaillée des raisons pour lesquelles l'opérateur de division a été modifié dans Python 3 et pourquoi l'ancienne division devrait être évitée.

Voir le sujet *Mathématiques simples* pour en savoir plus sur la division.

Réduire n'est plus un intégré

Dans Python 2, la `reduce` est disponible soit en tant que fonction intégrée, soit à partir du paquet `functools` (version 2.6 et `functools`), tandis qu'en Python 3, la `reduce` n'est disponible qu'à partir de `functools` . Cependant, la syntaxe pour `reduce` à la fois dans Python2 et Python3 est identique et est `reduce(function_to_reduce, list_to_reduce)` .

Par exemple, considérons la réduction d'une liste à une valeur unique en divisant chacun des nombres adjacents. Ici, nous utilisons la fonction `truediv` de la bibliothèque d' `operator` .

Dans Python 2.x, c'est aussi simple que:

Python 2.x 2.3

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

Dans Python 3.x, l'exemple devient un peu plus compliqué:

Python 3.x 3.0

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

Nous pouvons également utiliser à `from functools import reduce` pour éviter d'appeler `reduce` avec le nom de l'espace de noms.

Différences entre les fonctions range et xrange

Dans Python 2, la fonction `range` renvoie une liste, tandis que `xrange` crée un objet `xrange` spécial, qui est une séquence immuable, qui, contrairement à d'autres types de séquence intégrés, ne prend pas en charge les méthodes d' `index` ni de `count` :

Python 2.x 2.3

```
print(range(1, 10))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(isinstance(range(1, 10), list))
# Out: True

print(xrange(1, 10))
# Out: xrange(1, 10)

print(isinstance(xrange(1, 10), xrange))
# Out: True
```

En Python 3, `xrange` a été étendu à la `range` séquence, ce qui crée ainsi maintenant une `range` objet. Il n'y a pas de type de `xrange` :

Python 3.x 3.0

```
print(range(1, 10))
# Out: range(1, 10)

print(isinstance(range(1, 10), range))
```

```
# Out: True

# print(xrange(1, 10))
# The output will be:
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#NameError: name 'xrange' is not defined
```

De plus, depuis Python 3.2, la `range` prend également en charge le découpage, l' `index` et le `count` :

```
print(range(1, 10)[3:7])
# Out: range(3, 7)
print(range(1, 10).count(5))
# Out: 1
print(range(1, 10).index(7))
# Out: 6
```

L'avantage d'utiliser un type de séquence spécial au lieu d'une liste est que l'interpréteur n'a pas besoin d'allouer de la mémoire pour une liste et de la remplir:

Python 2.x 2.3

```
# range(10000000000000000)
# The output would be:
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# MemoryError

print(xrange(100000000000000000))
# Out: xrange(100000000000000000)
```

Comme ce dernier comportement est généralement souhaité, le premier a été supprimé dans Python 3. Si vous voulez toujours avoir une liste dans Python 3, vous pouvez simplement utiliser le constructeur `list()` sur un objet de `range` :

Python 3.x 3.0

```
print(list(range(1, 10)))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Compatibilité

Afin de maintenir la compatibilité entre les deux versions 2.x Python et Python 3.x, vous pouvez utiliser le `builtins` module du package externe `future` pour atteindre à la fois en *avant* et en *arrière-compatiblité-compatiblité:*

Python 2.x 2.0

```
#forward-compatible
from builtins import range
```

```
for i in range(10**8):
    pass
```

Python 3.x 3.0

```
#backward-compatible
from past.builtins import xrange

for i in xrange(10**8):
    pass
```

La `range` de la `future` bibliothèque prend en charge le découpage, l' `index` et le `count` dans toutes les versions de Python, tout comme la méthode intégrée sur Python 3.2+.

Déballer les Iterables

Python 3.x 3.0

Dans Python 3, vous pouvez décompresser une itération sans connaître le nombre exact d'éléments, et même avoir une variable contenant la fin de l'itérable. Pour cela, vous fournissez une variable pouvant collecter une liste de valeurs. Cela se fait en plaçant un astérisque avant le nom. Par exemple, décompresser une `list` :

```
first, second, *tail, last = [1, 2, 3, 4, 5]
print(first)
# Out: 1
print(second)
# Out: 2
print(tail)
# Out: [3, 4]
print(last)
# Out: 5
```

Remarque : Lorsque vous utilisez la syntaxe de la `*variable` , la `variable` sera toujours une liste, même si le type d'origine n'était pas une liste. Il peut contenir zéro ou plusieurs éléments en fonction du nombre d'éléments de la liste d'origine.

```
first, second, *tail, last = [1, 2, 3, 4]
print(tail)
# Out: [3]

first, second, *tail, last = [1, 2, 3]
print(tail)
# Out: []
print(last)
# Out: 3
```

De même, décompresser un `str` :

```
begin, *tail = "Hello"
print(begin)
# Out: 'H'
print(tail)
```

```
# Out: ['e', 'l', 'l', 'o']
```

Exemple de déballage d'une `date` ; `_` est utilisé dans cet exemple comme une variable jetable (nous nous intéressons uniquement à la valeur de l' `year`):

```
person = ('John', 'Doe', (10, 16, 2016))
*_, (*_, year_of_birth) = person
print(year_of_birth)
# Out: 2016
```

Il convient de mentionner que, puisque `*` mange un nombre variable d'éléments, vous ne pouvez pas avoir deux `*` *pour la même itération* dans une affectation - il ne saurait pas combien d'éléments entrent dans la première décompression, et combien dans la seconde. :

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
```

Python 3.x 3.5

Jusqu'à présent, nous avons discuté du déballage dans les missions. `*` et `**` ont été étendus dans Python 3.5 . Il est maintenant possible d'avoir plusieurs opérations de décompression dans une expression:

```
{*range(4), 4, *(5, 6, 7)}
# Out: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x 2.0

Il est également possible de décompresser une itération en arguments de fonction:

```
iterable = [1, 2, 3, 4, 5]
print(iterable)
# Out: [1, 2, 3, 4, 5]
print(*iterable)
# Out: 1 2 3 4 5
```

Python 3.x 3.5

Le déballage d'un dictionnaire utilise deux étoiles adjacentes `**` (PEP 448):

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
 # Out: {'x': 1, 'y': 2, 'z': 3}
```

Cela permet à la fois de remplacer les anciennes valeurs et de fusionner les dictionnaires.

```
dict1 = {'x': 1, 'y': 1}
dict2 = {'y': 2, 'z': 3}
{**dict1, **dict2}
# Out: {'x': 1, 'y': 2, 'z': 3}
```

Python 3.x 3.0

Python 3 a supprimé le déballage des tuple dans les fonctions. Par conséquent, ce qui suit ne fonctionne pas en Python 3

```
# Works in Python 2, but syntax error in Python 3:
map(lambda (x, y): x + y, zip(range(5), range(5)))
# Same is true for non-lambdas:
def example((x, y)):
    pass

# Works in both Python 2 and Python 3:
map(lambda x: x[0] + x[1], zip(range(5), range(5)))
# And non-lambdas, too:
def working_example(x_y):
    x, y = x_y
    pass
```

Voir PEP 3113 pour une justification détaillée.

Relever et gérer les exceptions

Voici la syntaxe Python 2, notez les virgules `,` sur les lignes `raise` et `except` :

Python 2.x 2.3

```
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

En Python 3, la `,` la syntaxe est supprimée et remplacée par des parenthèses et `as` mot - clé:

```
try:
    raise IOError("input/output error")
except IOError as exc:
    print(exc)
```

Pour assurer la compatibilité avec les versions antérieures, la syntaxe Python 3 est également disponible dans Python 2.6 et doit donc être utilisée pour tout nouveau code qui ne doit pas nécessairement être compatible avec les versions précédentes.

Python 3.x 3.0

Python 3 ajoute également le chaînage des exceptions , dans lequel vous pouvez signaler qu'une autre exception est à l' *origine* de cette exception. Par exemple

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from e
```

L'exception déclenchée dans l'instruction `except` est de type `DatabaseError` , mais l'exception d'origine est marquée comme attribut `__cause__` de cette exception. Lorsque la traceback est

affichée, l'exception d'origine sera également affichée dans la traceback:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Si vous lancez un bloc `except` *sans* chaînage explicite:

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}')
```

La traceback est

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Python 2.x 2.0

Ni l'un ni l'autre n'est pris en charge dans Python 2.x; l'exception d'origine et sa traceback seront perdues si une autre exception est déclenchée dans le bloc sauf. Le code suivant peut être utilisé pour la compatibilité:

```
import sys
import traceback

try:
    funcWithError()
except:
    sys_vers = getattr(sys, 'version_info', (0,))
    if sys_vers < (3, 0):
        traceback.print_exc()
    raise Exception("new exception")
```

Python 3.x 3.3

Pour "oublier" l'exception précédemment levée, utilisez `raise from None`

```
try:
    file = open('database.db')
except FileNotFoundError as e:
```

```
    raise DatabaseError('Cannot open {}') from None
```

Maintenant, le traceur serait simplement

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

Ou pour le rendre compatible avec Python 2 et 3, vous pouvez utiliser les six paquets comme suit:

```
import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('Cannot open {}'), None)
```

Méthode .next () sur les itérateurs renommés

Dans Python 2, un itérateur peut être parcouru en utilisant une méthode appelée `next` sur l'itérateur lui-même:

Python 2.x 2.3

```
g = (i for i in range(0, 3))
g.next()  # Yields 0
g.next()  # Yields 1
g.next()  # Yields 2
```

Dans Python 3, la méthode `.next` a été renommée en `.__next__` , reconnaissant son rôle «magic», l'appel de `.next` déclenchera donc une `.next AttributeError` . La manière correcte d'accéder à cette fonctionnalité dans Python 2 et Python 3 consiste à appeler la *fonction* `next` avec l'itérateur comme argument.

Python 3.x 3.0

```
g = (i for i in range(0, 3))
next(g)  # Yields 0
next(g)  # Yields 1
next(g)  # Yields 2
```

Ce code est portable entre les versions 2.6 et actuelles.

Comparaison de différents types

Python 2.x 2.3

Des objets de différents types peuvent être comparés. Les résultats sont arbitraires mais cohérents. Ils sont classés de manière à ce que `None` soit inférieur à tout, les types numériques sont plus petits que les types non numériques et tout le reste est ordonné par type lexicographiquement. Ainsi, un `int` est inférieur à un `str` et un `tuple` est supérieur à une `list` :

```
[1, 2] > 'foo'
# Out: False
(1, 2) > 'foo'
# Out: True
[1, 2] > (1, 2)
# Out: False
100 < [1, 'x'] < 'xyz' < (1, 'x')
# Out: True
```

Cela a été fait à l'origine afin qu'une liste de types mixtes puisse être triée et que les objets soient regroupés par type:

```
l = [7, 'x', (1, 2), [5, 6], 5, 8.0, 'y', 1.2, [7, 8], 'z']
sorted(l)
# Out: [1.2, 5, 7, 8.0, [5, 6], [7, 8], 'x', 'y', 'z', (1, 2)]
```

Python 3.x 3.0

Une exception est déclenchée lors de la comparaison de différents types (non numériques):

```
1 < 1.5
# Out: True

[1, 2] > 'foo'
# TypeError: unorderable types: list() > str()
(1, 2) > 'foo'
# TypeError: unorderable types: tuple() > str()
[1, 2] > (1, 2)
# TypeError: unorderable types: list() > tuple()
```

Pour trier les listes mixtes dans Python 3 par types et pour assurer la compatibilité entre les versions, vous devez fournir une clé pour la fonction triée:

```
>>> list = [1, 'hello', [3, 4], {'python': 2}, 'stackoverflow', 8, {'python': 3}, [5, 6]]
>>> sorted(list, key=str)
# Out: [1, 8, [3, 4], [5, 6], 'hello', 'stackoverflow', {'python': 2}, {'python': 3}]
```

L'utilisation de `str` comme fonction `key` convertit temporairement chaque élément en chaîne uniquement à des fins de comparaison. Il voit ensuite la représentation des chaînes commençant par `[` , `'` , `{` ou `0-9` et il est capable de les trier (et tous les caractères suivants).

Entrée utilisateur

En Python 2, les entrées utilisateur sont acceptées en utilisant la fonction `raw_input` ,

Python 2.x 2.3

```
user_input = raw_input()
```

En Python 3, l'entrée utilisateur est acceptée à l'aide de la fonction d' `input` .

Python 3.x 3.0

```
user_input = input()
```

En Python 2, l' `input` fonction accepte l' entrée et l' *interpréter.* Bien que cela puisse être utile, il a plusieurs considérations de sécurité et a été supprimé dans Python 3. Pour accéder à la même fonctionnalité, `eval(input())` peut être utilisé.

Pour garder un script portable entre les deux versions, vous pouvez placer le code ci-dessous en haut de votre script Python:

```
try:
    input = raw_input
except NameError:
    pass
```

Changement de méthode de dictionnaire

Dans Python 3, de nombreuses méthodes de dictionnaire ont un comportement très différent de celui de Python 2, et beaucoup ont été supprimées également: `has_key` , `iter*` et `view*` ont disparu. Au lieu de `d.has_key(key)` , longtemps obsolète, il faut maintenant utiliser `key in d` .

Dans Python 2, les `keys` méthodes de dictionnaire, les `values` et les `items` renvoient des listes. Dans Python 3, ils renvoient des objets de *vue à la* place; les objets de vue ne sont pas des itérateurs et ils en diffèrent de deux manières, à savoir:

- ils ont la taille (on peut utiliser la fonction `len` sur eux)
- ils peuvent être répétés plusieurs fois

De plus, comme avec les itérateurs, les modifications apportées au dictionnaire sont reflétées dans les objets de vue.

Python 2.7 a rétrogradé ces méthodes depuis Python 3; Ils sont disponibles sous `viewkeys` de `viewkeys` , de `viewvalues` et de `viewitems` . Pour transformer le code Python 2 en code Python 3, les formulaires correspondants sont les suivants:

- `d.keys()` , `d.values()` et `d.items()` de Python 2 doivent être remplacés par `list(d.keys())` , `list(d.values())` et `list(d.items())`
- `d.iterkeys()` , `d.itervalues()` et `d.iteritems()` doivent être changés en `iter(d.keys())` , ou même mieux, `iter(d)` ; `iter(d.values())` et `iter(d.items())` respectivement
- et enfin les méthodes `d.viewkeys()` , `d.viewvalues()` et `d.viewitems()` peuvent être remplacées par `d.keys()` , `d.values()` et `d.items()` .

Le portage du code Python 2 qui *itère* sur les clés, les valeurs ou les éléments du dictionnaire tout en le mutant est parfois délicat. Considérer:

```
d = {'a': 0, 'b': 1, 'c': 2, '!': 3}
for key in d.keys():
    if key.isalpha():
        del d[key]
```

Le code semble fonctionner de la même manière dans Python 3, mais la méthode `keys` renvoie un objet de vue, pas une liste, et si le dictionnaire change de taille en itération, le code Python 3 se bloquera avec `RuntimeError: dictionary changed size during iteration` . La solution est bien sûr d'écrire correctement `for key in list(d)` .

De même, les objets de vue se comportent différemment des itérateurs: on ne peut pas utiliser `next()` et on ne peut pas *reprendre l'* itération; au lieu de cela redémarrer; Si le code Python 2 transmet la valeur de retour de `d.iterkeys()` , `d.itervalues()` ou `d.iteritems()` à une méthode qui attend un itérateur au lieu d'une *itération* , cela devrait être `iter(d)` , `iter(d.values())` ou `iter(d.items())` dans Python 3.

instruction exec est une fonction dans Python 3

Dans Python 2, `exec` est une instruction, avec une syntaxe spéciale: `exec code [in globals[, locals]]` . Dans Python 3, `exec` est maintenant une fonction: `exec(code, [, globals[, locals]])` , et la syntaxe Python 2 `SyntaxError` une `SyntaxError` .

Au fur et à mesure que `print` passait d'une instruction à une fonction, une importation `__future__` était également ajoutée. Cependant, il n'y a pas `from __future__ import exec_function` , car il n'est pas nécessaire: l'instruction exec dans Python 2 peut également être utilisée avec une syntaxe qui ressemble exactement à l' `exec` fonction `exec` dans Python 3. Vous pouvez donc modifier les instructions

Python 2.x 2.3

```
exec 'code'
exec 'code' in global_vars
exec 'code' in global_vars, local_vars
```

aux formes

Python 3.x 3.0

```
exec('code')
exec('code', global_vars)
exec('code', global_vars, local_vars)
```

et ces dernières formes sont garanties de fonctionner de manière identique dans Python 2 et Python 3.

bug de la fonction hasattr dans Python 2

Dans Python 2, lorsqu'une propriété `hasattr` une erreur, `hasattr` ignore cette propriété et renvoie `False` .

```
class A(object):
    @property
    def get(self):
        raise IOError
```

```
class B(object):
    @property
    def get(self):
        return 'get in b'

a = A()
b = B()

print 'a hasattr get: ', hasattr(a, 'get')
# output False in Python 2 (fixed, True in Python 3)
print 'b hasattr get', hasattr(b, 'get')
# output True in Python 2 and Python 3
```

Ce bug est corrigé dans Python3. Donc, si vous utilisez Python 2, utilisez

```
try:
    a.get
except AttributeError:
    print("no get property!")
```

ou utilisez plutôt `getattr`

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("no get property!")
```

Modules renommés

Quelques modules de la bibliothèque standard ont été renommés:

Ancien nom	Nouveau nom
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
_markupbase	markupbase
repr	reproche
test.test_support	test.support
Tkinter	tkinter

Ancien nom	Nouveau nom
tkFileDialog	tkinter.filedialog
urllib / urllib2	urllib, urllib.parse, urllib.error, urllib.response, urllib.request, urllib.robotparser

Certains modules ont même été convertis de fichiers en bibliothèques. Prenons l'exemple de tkinter et urllib ci-dessus.

Compatibilité

Lorsque vous maintenez la compatibilité entre les versions Python 2.x et 3.x, vous pouvez utiliser le `future` package externe pour importer des packages de bibliothèque standard de niveau supérieur avec des noms Python 3.x sur les versions Python 2.x.

Constantes Octales

Dans Python 2, un littéral octal pourrait être défini comme

```
>>> 0755  # only Python 2
```

Pour assurer la compatibilité croisée, utilisez

```
0o755  # both Python 2 and Python 3
```

Toutes les classes sont des "nouvelles classes" dans Python 3.

Dans Python `3.x` toutes les classes sont des classes de *nouveau style* ; lors de la définition d'une nouvelle classe, python le fait hériter implicitement de l' `object` . En tant que tel, la spécification d'un `object` dans une définition de `class` est une option complètement facultative:

Python 3.x 3.0

```
class X: pass
class Y(object): pass
```

Ces deux classes contiennent désormais des `object` dans leur `mro` (ordre de résolution de la méthode):

Python 3.x 3.0

```
>>> X.__mro__
(__main__.X, object)

>>> Y.__mro__
(__main__.Y, object)
```

Dans Python `2.x` classes sont, par défaut, d'anciennes classes; ils n'héritent pas implicitement d' `object` . Cela fait que la sémantique des classes diffère selon que l'on ajoute explicitement un `object` tant que `class` base:

Python 2.x 2.3

```
class X: pass
class Y(object): pass
```

Dans ce cas, si nous essayons d'imprimer le `__mro__` de `Y` , une sortie similaire à celle du cas Python `3.x` apparaîtra:

Python 2.x 2.3

```
>>> Y.__mro__
(<class '__main__.Y'>, <type 'object'>)
```

Cela se produit parce que nous avons explicitement fait hériter `Y` de l'objet lors de sa définition: `class Y(object): pass` . Pour la classe `X` qui n'hérite *pas* d'objet, l'attribut `__mro__` n'existe pas, en essayant d'y accéder aboutit à une erreur `AttributeError` .

Afin de **garantir la compatibilité** entre les deux versions de Python, les classes peuvent être définies avec un `object` comme classe de base:

```
class mycls(object):
    """I am fully compatible with Python 2/3"""
```

Alternativement, si la variable `__metaclass__` est définie pour `type` une portée globale, toutes les classes définies ultérieurement dans un module donné sont implicitement new-style sans avoir à hériter explicitement de l' `object` :

```
__metaclass__ = type

class mycls:
    """I am also fully compatible with Python 2/3"""
```

Suppression des opérateurs <> et ``, synonyme de! = Et repr ()

Dans Python 2, `<>` est un synonyme de `!=` ; de même, `` `foo` `` est un synonyme de `repr(foo)` .

Python 2.x 2.7

```
>>> 1 <> 2
True
>>> 1 <> 1
False
>>> foo = 'hello world'
>>> repr(foo)
"'hello world'"
>>> `foo`
"'hello world'"
```

Python 3.x 3.0

```
>>> 1 <> 2
  File "<stdin>", line 1
    1 <> 2
       ^
SyntaxError: invalid syntax
>>> `foo`
  File "<stdin>", line 1
    `foo`
    ^
SyntaxError: invalid syntax
```

encoder / décoder en hexadécimal n'est plus disponible

Python 2.x 2.7

```
"1deadbeef3".decode('hex')
# Out: '\x1d\xea\xdb\xee\xf3'
'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Out: 1deadbeef3
```

Python 3.x 3.0

```
"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: 'str' object has no attribute 'decode'

b"1deadbeef3".decode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.decode() to handle arbitrary codecs

'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.encode() to handle arbitrary codecs

b'\x1d\xea\xdb\xee\xf3'.encode('hex')
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# AttributeError: 'bytes' object has no attribute 'encode'
```

Cependant, comme suggéré par le message d'erreur, vous pouvez utiliser le module de `codecs` pour obtenir le même résultat:

```
import codecs
codecs.decode('1deadbeef4', 'hex')
# Out: b'\x1d\xea\xdb\xee\xf4'
codecs.encode(b'\x1d\xea\xdb\xee\xf4', 'hex')
# Out: b'1deadbeef4'
```

Notez que `codecs.encode` renvoie un objet `bytes` . Pour obtenir un objet `str` , il suffit de `decode` en ASCII:

```
codecs.encode(b'\x1d\xea\xdb\xee\xff', 'hex').decode('ascii')
# Out: '1deadbeeff'
```

Fonction cmp supprimée dans Python 3

Dans Python 3, la fonction intégrée `cmp` été supprimée, avec la méthode spéciale `__cmp__` .

De la documentation:

> La fonction `cmp()` doit être considérée comme disparue et la méthode spéciale `__cmp__()` n'est plus prise en charge. Utilisez `__lt__()` pour le tri, `__eq__()` avec `__hash__()` , et d'autres comparaisons riches si nécessaire. (Si vous avez vraiment besoin de la fonctionnalité `cmp()` , vous pouvez utiliser l'expression `(a > b) - (a < b)` comme équivalent pour `cmp(a, b)` .)

De plus, toutes les fonctions intégrées acceptant le paramètre `cmp` acceptent désormais uniquement le paramètre `key` mot clé uniquement.

Dans le module `functools` il y a aussi une fonction utile `cmp_to_key(func)` qui vous permet de convertir une fonction de style `cmp` une fonction de style `key` :

> Transformer une ancienne fonction de comparaison en une fonction clé. Utilisé avec les outils qui acceptent les fonctions clés (telles que `sorted()` , `min()` , `max()` , `heapq.nlargest()` , `heapq.nsmallest()` , `itertools.groupby()`). Cette fonction est principalement utilisée comme outil de transition pour les programmes en cours de conversion à partir de Python 2 qui prend en charge l'utilisation de fonctions de comparaison.

Variables fuites dans la compréhension de la liste

Python 2.x 2.3

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'U'
```

Python 3.x 3.0

```
x = 'hello world!'
vowels = [x for x in 'AEIOU']

print (vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'hello world!'
```

Comme on peut le voir sur l'exemple, dans Python 2, la valeur de `x` été divulguée: cela masquait

`hello world!` et imprimé `U` , puisque c'était la dernière valeur de `x` à la fin de la boucle.

Cependant, en Python 3 `x` imprime le monde original défini à l'origine `hello world!` , puisque la variable locale de la compréhension de la liste ne masque pas les variables de la portée environnante.

De plus, ni les expressions génératrices (disponibles dans Python depuis la version 2.5), ni les interprétations de dictionnaire ou de jeu (qui ont été renvoyées vers Python 2.7 à partir de Python 3), ne contenaient de variables dans Python 2.

Notez que dans Python 2 et Python 3, les variables entrent dans la portée environnante lors de l'utilisation d'une boucle for:

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
print(x)
# Out: 'U'
```

carte()

`map()` est une méthode intégrée utile pour appliquer une fonction aux éléments d'une itération. Dans Python 2, `map` renvoie une liste. Dans Python 3, `map` renvoie un *objet map* , qui est un générateur.

```
# Python 2.X
>>> map(str, [1, 2, 3, 4, 5])
['1', '2', '3', '4', '5']
>>> type(_)
>>> <class 'list'>

# Python 3.X
>>> map(str, [1, 2, 3, 4, 5])
<map object at 0x*>
>>> type(_)
<class 'map'>

# We need to apply map again because we "consumed" the previous map....
>>> map(str, [1, 2, 3, 4, 5])
>>> list(_)
['1', '2', '3', '4', '5']
```

Dans Python 2, vous pouvez transmettre `None` pour servir de fonction d'identité. Cela ne fonctionne plus dans Python 3.

Python 2.x 2.3

```
>>> map(None, [0, 1, 2, 3, 0, 4])
[0, 1, 2, 3, 0, 4]
```

Python 3.x 3.0

```
>>> list(map(None, [0, 1, 2, 3, 0, 5]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

De plus, lors du passage de plusieurs arguments itérables en tant que arguments dans Python 2, les plaquettes de `map` les itérables les plus courtes avec `None` (similaire à `itertools.izip_longest`). Dans Python 3, l'itération s'arrête après la plus courte itération.

En Python 2:

Python 2.x 2.3

```
>>> map(None, [1, 2, 3], [1, 2], [1, 2, 3, 4, 5])
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

En Python 3:

Python 3.x 3.0

```
>>> list(map(lambda x, y, z: (x, y, z), [1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2)]

# to obtain the same padding as in Python 2 use zip_longest from itertools
>>> import itertools
>>> list(itertools.zip_longest([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

Note : au lieu de `map` utilisez des listes compréhensibles compatibles avec Python 2/3. Remplacement de la `map(str, [1, 2, 3, 4, 5])` :

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

filter (), map () et zip () renvoient des itérateurs au lieu de séquences

Python 2.x 2.7

Dans le `filter` Python 2, les fonctions intégrées `map` et `zip` renvoient une séquence. `map` et `zip` renvoient toujours une liste alors qu'avec `filter` le type de retour dépend du type de paramètre donné:

```
>>> s = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> s
'abc'
>>> s = map(lambda x: x * x, [0, 1, 2])
>>> s
[0, 1, 4]
>>> s = zip([0, 1, 2], [3, 4, 5])
>>> s
[(0, 3), (1, 4), (2, 5)]
```

Python 3.x 3.0

Dans Python 3 `filter` , `map` et `zip` return iterator à la place:

```
>>> it = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> it
<filter object at 0x00000098A55C2518>
>>> ''.join(it)
'abc'
>>> it = map(lambda x: x * x, [0, 1, 2])
>>> it
<map object at 0x000000E0763C2D30>
>>> list(it)
[0, 1, 4]
>>> it = zip([0, 1, 2], [3, 4, 5])
>>> it
<zip object at 0x000000E0763C52C8>
>>> list(it)
[(0, 3), (1, 4), (2, 5)]
```

Puisque Python 2 `itertools.izip` est équivalent à Python 3, `zip` `izip` a été supprimé sur Python 3.

Importations absolues / relatives

Dans Python 3, PEP 404 modifie la façon dont les importations fonctionnent à partir de Python 2. *Les* importations *relatives implicites* ne sont plus autorisées dans les packages et à `from ... import *` importations ne sont autorisées que dans le code de niveau module.

Pour obtenir le comportement de Python 3 dans Python 2:

- la fonctionnalité d' importation absolue peut être activée à `from __future__ import absolute_import`
- *les* importations *relatives explicites* sont encouragées à la place des importations *implicites relatives*

Pour plus de clarté, dans Python 2, un module peut importer le contenu d'un autre module situé dans le même répertoire, comme suit:

```
import foo
```

Notez que l'emplacement de `foo` est ambigu à partir de la seule déclaration d'importation. Ce type d'importation implicite relative est donc déconseillé en faveur d' importations relatives explicites , qui ressemblent à ce qui suit:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
from ...package import bar
from ...sys import path
```

Le point `.` permet une déclaration explicite de l'emplacement du module dans l'arborescence.

Plus sur les importations relatives

Considérons un paquet défini par l'utilisateur appelé `shapes` . La structure du répertoire est la suivante:

```
shapes
├── __init__.py
|
├── circle.py
|
├── square.py
|
└── triangle.py
```

`circle.py` , `square.py` et `triangle.py` importent tous `util.py` tant que module. Comment vont-ils se référer à un module du même niveau?

```
from . import util # use util.PI, util.sq(x), etc
```

OU

```
from .util import * #use PI, sq(x), etc to call functions
```

Le `.` est utilisé pour les importations relatives de même niveau.

Considérons maintenant une autre disposition du module de `shapes` :

```
shapes
├── __init__.py
|
├── circle
|   ├── __init__.py
|   └── circle.py
|
├── square
|   ├── __init__.py
|   └── square.py
|
├── triangle
|   ├── __init__.py
|   ├── triangle.py
|
└── util.py
```

Maintenant, comment ces 3 classes se réfèrent-elles à util.py?

```
from .. import util # use util.PI, util.sq(x), etc
```

OU

```
from ..util import * # use PI, sq(x), etc to call functions
```

Le `..` est utilisé pour les importations relatives au niveau parent. Ajouter plus `.` s avec nombre de niveaux entre le parent et l'enfant.

Fichier I / O

`file` n'est plus un nom intégré dans 3.x (`open` works).

Les détails internes des E / S de fichiers ont été déplacés vers le module `io` standard de la bibliothèque, qui est également la nouvelle `StringIO` de `StringIO` :

```
import io
assert io.open is open # the builtin is an alias
buffer = io.StringIO()
buffer.write('hello, ') # returns number of characters written
buffer.write('world!\n')
buffer.getvalue() # 'hello, world!\n'
```

Le mode fichier (text vs binary) détermine maintenant le type de données produit en lisant un fichier (et le type requis pour l'écriture):

```
with open('data.txt') as f:
    first_line = next(f)
    assert type(first_line) is str
with open('data.bin', 'rb') as f:
    first_kb = f.read(1024)
    assert type(first_kb) is bytes
```

Le codage des fichiers texte est `locale.getpreferredencoding(False)` par défaut sur tout ce qui est renvoyé par `locale.getpreferredencoding(False)` . Pour spécifier explicitement un codage, utilisez le paramètre de mot-clé `encoding` :

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

La fonction round () et le type de retour

bris de cravate

En Python 2, l'utilisation de `round()` sur un nombre proche de deux nombres entiers renvoie la valeur la plus éloignée de 0. Par exemple:

Python 2.x 2.7

```
round(1.5)  # Out: 2.0
round(0.5)  # Out: 1.0
round(-0.5)  # Out: -1.0
round(-1.5)  # Out: -2.0
```

Dans Python 3 cependant, `round()` renverra l'entier pair (*l'arrondi des banquiers*). Par exemple:

Python 3.x 3.0

```
round(1.5)  # Out: 2
round(0.5)  # Out: 0
round(-0.5)  # Out: 0
round(-1.5)  # Out: -2
```

La fonction round () suit la stratégie d' *arrondissement de moitié à pair qui arrondira les* nombres à mi-chemin au nombre entier pair le plus proche (par exemple, `round(2.5)` renvoie désormais 2 plutôt que 3.0).

Selon la référence dans Wikipedia , ceci est également connu sous le nom d' *arrondi sans biais, d'arrondi convergent, d'arrondi de statisticien, d'arrondissement hollandais, d'arrondi gaussien* ou d' *arrondi impair* .

La moitié à l'arrondissement fait partie de la norme IEEE 754 et c'est aussi le mode d'arrondi par défaut dans Microsoft .NET.

Cette stratégie d'arrondissement tend à réduire l'erreur d'arrondi totale. Étant donné qu'en moyenne, le nombre de chiffres arrondis est le même que le nombre de chiffres arrondis, les erreurs d'arrondi sont annulées. D'autres méthodes d'arrondi ont plutôt tendance à avoir un biais à la hausse ou à la baisse dans l'erreur moyenne.

---

type de retour round ()

La fonction `round()` renvoie un type `float` dans Python 2.7

Python 2.x 2.7

```
round(4.8)
# 5.0
```

À partir de Python 3.0, si le second argument (nombre de chiffres) est omis, il renvoie un `int` .

Python 3.x 3.0

```
round(4.8)
# 5
```

Vrai, Faux et Aucun

Dans Python 2, `True` , `False` et `None` sont des constantes intégrées. Ce qui signifie qu'il est possible de les réaffecter.

Python 2.x 2.0

```
True, False = False, True
True   # False
False  # True
```

Vous ne pouvez pas faire cela avec `None` depuis Python 2.4.

Python 2.x 2.4

```
None = None  # SyntaxError: cannot assign to None
```

Dans Python 3, `True` , `False` et `None` sont maintenant des mots-clés.

Python 3.x 3.0

```
True, False = False, True  # SyntaxError: can't assign to keyword

None = None  # SyntaxError: can't assign to keyword
```

Renvoie la valeur lors de l'écriture dans un objet fichier

En Python 2, l'écriture directe dans un descripteur de fichier renvoie `None` :

Python 2.x 2.3

```
hi = sys.stdout.write('hello world\n')
# Out: hello world
type(hi)
# Out: <type 'NoneType'>
```

Dans Python 3, l'écriture dans un handle renvoie le nombre de caractères écrits lors de l'écriture du texte et le nombre d'octets écrits lors de l'écriture des octets:

Python 3.x 3.0

```
import sys

char_count = sys.stdout.write('hello world \n')
# Out: hello world 
char_count
# Out: 14

byte_count = sys.stdout.buffer.write(b'hello world \xf0\x9f\x90\x8d\n')
# Out: hello world 
byte_count
# Out: 17
```

long vs int

Dans Python 2, tout entier supérieur à C `ssize_t` serait converti en type de données `long` , indiqué par un suffixe `L` sur le littéral. Par exemple, sur une version 32 bits de Python:

Python 2.x 2.7

```
>>> 2**31
2147483648L
>>> type(2**31)
<type 'long'>
>>> 2**30
1073741824
>>> type(2**30)
```

```
<type 'int'>
>>> 2**31 - 1  # 2**31 is long and long - int is long
2147483647L
```

Cependant, dans Python 3, le type de données `long` été supprimé. peu importe la taille de l'entier, ce sera un `int` .

Python 3.x 3.0

```
2**1024
# Output:
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021

print(-(2**1024))
# Output: -
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021

type(2**1024)
# Output: <class 'int'>
```

Valeur booléenne de classe

Python 2.x 2.7

Dans Python 2, si vous souhaitez définir vous-même une valeur booléenne de classe, vous devez implémenter la méthode `__nonzero__` dans votre classe. La valeur est True par défaut.

```
class MyClass:
    def __nonzero__(self):
        return False

my_instance = MyClass()
print bool(MyClass)        # True
print bool(my_instance)    # False
```

Python 3.x 3.0

En Python 3, `__bool__` est utilisé à la place de `__nonzero__`

```
class MyClass:
    def __bool__(self):
        return False

my_instance = MyClass()
print(bool(MyClass))        # True
print(bool(my_instance))    # False
```

Lire Incompatibilités entre Python 2 et Python 3 en ligne:
https://riptutorial.com/fr/python/topic/809/incompatibilites-entre-python-2-et-python-3

Chapitre 89: Indexation et découpage

Syntaxe

- obj [start: stop: step]
- tranche
- tranche (démarrer, arrêter [, étape])

Paramètres

Paramètre	La description
`obj`	L'objet que vous voulez extraire d'un "sous-objet" de
`start`	L'index d' `obj` le sous-objet doit commencer (gardez à l'esprit que Python est indexé à zéro, ce qui signifie que le premier élément d' `obj` a un index de `0`). Si omis, la valeur par défaut est `0` .
`stop`	L'index (non inclusif) d' `obj` auquel vous souhaitez que le sous-objet se termine. S'il est omis, le paramètre par défaut est `len(obj)` .
`step`	Vous permet de sélectionner uniquement chaque élément de l' `step` . Si omis, la valeur par défaut est `1` .

Remarques

Vous pouvez unifier le concept de découpage en tranches avec celui de découper d'autres séquences en visualisant des chaînes sous la forme d'une collection immuable de caractères, en sachant qu'un caractère unicode est représenté par une chaîne de longueur 1.

En notation mathématique, vous pouvez envisager de trancher pour utiliser un intervalle à moitié ouvert de `[start, end)` , c'est-à-dire que le début est inclus, mais que la fin ne l'est pas. Le caractère semi-ouvert de l'intervalle a l'avantage que `len(x[:n])` = `n` où `len(x)` > = `n` , tandis que l'intervalle fermé au début a l'avantage que `x[n:n+1]` = `[x[n]]` où `x` est une liste avec `len(x) >= n` , conservant ainsi la cohérence entre la notation par indexation et celle par découpage.

Examples

Tranchage de base

Pour toute itération (par exemple, une chaîne, une liste, etc.), Python vous permet de découper et de retourner une sous-chaîne ou une sous-liste de ses données.

Format de découpage:

```
iterable_name[start:stop:step]
```

où,

- `start` est le premier index de la tranche. La valeur par défaut est 0 (l'index du premier élément)
- `stop` vous au-delà du dernier index de la tranche. La valeur par défaut est len (itérable)
- `step` est la taille de l'étape (mieux expliquée par les exemples ci-dessous)

Exemples:

```
a = "abcdef"
a             # "abcdef"
              # Same as a[:] or a[::] since it uses the defaults for all three indices
a[-1]         # "f"
a[:]          # "abcdef"
a[::]         # "abcdef"
a[3:]         # "def" (from index 3, to end(defaults to size of iterable))
a[:4]         # "abcd" (from beginning(default 0) to position 4 (excluded))
a[2:4]        # "cd" (from position 2, to position 4 (excluded))
```

En outre, l'un des éléments ci-dessus peut être utilisé avec la taille de pas définie:

```
a[::2]        # "ace" (every 2nd element)
a[1:4:2]      # "bd" (from index 1, to index 4 (excluded), every 2nd element)
```

Les indices peuvent être négatifs, auquel cas ils sont calculés à partir de la fin de la séquence

```
a[:-1]      # "abcde" (from index 0 (default), to the second last element (last element - 1))
a[:-2]      # "abcd" (from index 0 (default), to the third last element (last element -2))
a[-1:]      # "f" (from the last element to the end (default len())
```

Les tailles d'étape peuvent également être négatives, auquel cas la tranche parcourra la liste dans l'ordre inverse:

```
a[3:1:-1]   # "dc" (from index 2 to None (default), in reverse order)
```

Cette construction est utile pour inverser une itération

```
a[::-1]     # "fedcba" (from last element (default len()-1), to first, in reverse order(-1))
```

Notez que pour les étapes négatives, `end_index` par défaut est `None` (voir http://stackoverflow.com/a/12521981)

```
a[5:None:-1] # "fedcba" (this is equivalent to a[::-1])
a[5:0:-1]    # "fedcb" (from the last element (index 5) to second element (index 1)
```

Faire une copie superficielle d'un tableau

Un moyen rapide de faire une copie d'un tableau (par opposition à l'attribution d'une variable avec une autre référence au tableau d'origine) est la suivante:

```
arr[:]
```

Examinons la syntaxe. `[:]` signifie que `start` , `end` et `slice` sont tous omis. Ils sont par défaut à `0` , `len(arr)` et `1` , respectivement, ce qui signifie que le sous-tableau que nous demandons aura tous les éléments de `arr` du début à la fin.

En pratique, cela ressemble à quelque chose comme:

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)    # ['a', 'b', 'c', 'd']
print(copy)   # ['a', 'b', 'c']
```

Comme vous pouvez le voir, `arr.append('d')` ajouté `d` à `arr` , mais la `copy` est restée inchangée!

Notez que cela fait une copie *superficielle* et est identique à `arr.copy()` .

Inverser un objet

Vous pouvez utiliser des tranches pour inverser très facilement un `str` , une `list` ou un `tuple` (ou, fondamentalement, tout objet de collection qui implémente le découpage en tranches avec le paramètre step). Voici un exemple d'inversion d'une chaîne, bien que cela s'applique également aux autres types répertoriés ci-dessus:

```
s = 'reverse me!'
s[::-1]    # '!em esrever'
```

Regardons rapidement la syntaxe. `[::-1]` signifie que la tranche doit être du début jusqu'à la fin de la chaîne (parce que `start` et `end` sont omis) et une étape de `-1` signifie qu'elle doit se déplacer dans la chaîne en sens inverse.

Indexation des classes personnalisées: __getitem__, __setitem__ et __delitem__

```
class MultiIndexingList:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return repr(self.value)

    def __getitem__(self, item):
        if isinstance(item, (int, slice)):
            return self.__class__(self.value[item])
        return [self.value[i] for i in item]

    def __setitem__(self, item, value):
```

```
        if isinstance(item, int):
            self.value[item] = value
        elif isinstance(item, slice):
            raise ValueError('Cannot interpret slice with multiindexing')
        else:
            for i in item:
                if isinstance(i, slice):
                    raise ValueError('Cannot interpret slice with multiindexing')
                self.value[i] = value

    def __delitem__(self, item):
        if isinstance(item, int):
            del self.value[item]
        elif isinstance(item, slice):
            del self.value[item]
        else:
            if any(isinstance(elem, slice) for elem in item):
                raise ValueError('Cannot interpret slice with multiindexing')
            item = sorted(item, reverse=True)
            for elem in item:
                del self.value[elem]
```

Cela permet de trancher et d'indexer pour l'accès aux éléments:

```
a = MultiIndexingList([1,2,3,4,5,6,7,8])
a
# Out: [1, 2, 3, 4, 5, 6, 7, 8]
a[1,5,2,6,1]
# Out: [2, 6, 3, 7, 2]
a[4, 1, 5:, 2, ::2]
# Out: [5, 2, [6, 7, 8], 3, [1, 3, 5, 7]]
#       4|1-|----50:---|2-|-----::2-----   <-- indicated which element came from which index
```

Lorsque la définition et la suppression d'éléments ne permettent que l'indexation d'entiers *séparés* par des *virgules* (sans découpage):

```
a[4] = 1000
a
# Out: [1, 2, 3, 4, 1000, 6, 7, 8]
a[2,6,1] = 100
a
# Out: [1, 100, 100, 4, 1000, 6, 100, 8]
del a[5]
a
# Out: [1, 100, 100, 4, 1000, 100, 8]
del a[4,2,5]
a
# Out: [1, 100, 4, 8]
```

Assignation de tranche

Une autre fonctionnalité intéressante utilisant des tranches est l'affectation des tranches. Python vous permet d'affecter de nouvelles tranches pour remplacer les anciennes tranches d'une liste en une seule opération.

Cela signifie que si vous avez une liste, vous pouvez remplacer plusieurs membres dans une

seule tâche:

```
lst = [1, 2, 3]
lst[1:3] = [4, 5]
print(lst) # Out: [1, 4, 5]
```

L'affectation ne doit pas non plus correspondre à la taille, donc si vous voulez remplacer une ancienne tranche par une nouvelle tranche de taille différente, vous pouvez:

```
lst = [1, 2, 3, 4, 5]
lst[1:4] = [6]
print(lst) # Out: [1, 6, 5]
```

Il est également possible d'utiliser la syntaxe de découpage connue pour faire des choses comme remplacer la liste entière:

```
lst = [1, 2, 3]
lst[:] = [4, 5, 6]
print(lst) # Out: [4, 5, 6]
```

Ou seulement les deux derniers membres:

```
lst = [1, 2, 3]
lst[-2:] = [4, 5, 6]
print(lst) # Out: [1, 4, 5, 6]
```

Trancher des objets

Les tranches sont des objets en elles-mêmes et peuvent être stockées dans des variables avec la fonction `slice()` intégrée. Les variables Slice peuvent être utilisées pour rendre votre code plus lisible et pour promouvoir la réutilisation.

```
>>> programmer_1 = [ 1956, 'Guido', 'van Rossum', 'Python', 'Netherlands']
>>> programmer_2 = [ 1815, 'Ada', 'Lovelace', 'Analytical Engine', 'England']
>>> name_columns = slice(1, 3)
>>> programmer_1[name_columns]
['Guido', 'van Rossum']
>>> programmer_2[name_columns]
['Ada', 'Lovelace']
```

Indexation de base

Les listes Python sont basées sur 0, c'est-à- *dire que* le premier élément de la liste est accessible par l'index `0`

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

Vous pouvez accéder au deuxième élément de la liste par index `1` , troisième élément par index `2`

et ainsi de suite:

```
print(arr[1])
>> 'b'
print(arr[2])
>> 'c'
```

Vous pouvez également utiliser des index négatifs pour accéder aux éléments depuis la fin de la liste. par exemple. index `-1` vous donnera le dernier élément de la liste et l'index `-2` vous donnera l'avant-dernier élément de la liste:

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

Si vous essayez d'accéder à un index qui n'est pas présent dans la liste, une `IndexError` sera `IndexError` :

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Lire Indexation et découpage en ligne: https://riptutorial.com/fr/python/topic/289/indexation-et-decoupage

Chapitre 90: Interface de passerelle de serveur Web (WSGI)

Paramètres

Paramètre	Détails
start_response	Une fonction utilisée pour traiter le début

Examples

Objet serveur (méthode)

Notre objet serveur reçoit un paramètre "application" qui peut être n'importe quel objet d'application appelable (voir d'autres exemples). Il écrit d'abord les en-têtes, puis le corps des données renvoyées par notre application sur la sortie standard du système.

```
import os, sys

def run(application):
    environ['wsgi.input']        = sys.stdin
    environ['wsgi.errors']       = sys.stderr

    headers_set = []
    headers_sent = []

    def write (data):
        """
        Writes header data from 'start_response()' as well as body data from 'response'
        to system standard output.
        """
        if not headers_set:
            raise AssertionError("write() before start_response()")

        elif not headers_sent:
            status, response_headers = headers_sent[:] = headers_set
            sys.stdout.write('Status: %s\r\n' % status)
            for header in response_headers:
                sys.stdout.write('%s: %s\r\n' % header)
            sys.stdout.write('\r\n')

        sys.stdout.write(data)
        sys.stdout.flush()

    def start_response(status, response_headers):
        """ Sets headers for the response returned by this server."""
        if headers_set:
            raise AssertionError("Headers already set!")

        headers_set[:] = [status, response_headers]
        return write
```

```
    # This is the most important piece of the 'server object'
    # Our result will be generated by the 'application' given to this method as a parameter
    result = application(environ, start_response)
    try:
        for data in result:
            if data:
                write(data)          # Body isn't empty send its data to 'write()'
        if not headers_sent:
            write('')                # Body is empty, send empty string to 'write()'
```

Lire Interface de passerelle de serveur Web (WSGI) en ligne:
https://riptutorial.com/fr/python/topic/5315/interface-de-passerelle-de-serveur-web--wsgi-

Chapitre 91: Introduction à RabbitMQ en utilisant AMQPStorm

Remarques

La dernière version d' AMQPStorm est disponible sur pypi ou vous pouvez l'installer en utilisant pip

```
pip install amqpstorm
```

Examples

Comment consommer des messages de RabbitMQ

Commencez par importer la bibliothèque.

```
from amqpstorm import Connection
```

Lors de la consommation de messages, nous devons d'abord définir une fonction pour gérer les messages entrants. Cela peut être n'importe quelle fonction appelable et doit prendre un objet de message ou un tuple de message (en fonction du paramètre `to_tuple` défini dans `start_consuming`).

Outre le traitement des données du message entrant, nous devrons également accuser réception ou rejeter le message. Ceci est important, car nous devons informer RabbitMQ que nous avons correctement reçu et traité le message.

```
def on_message(message):
    """This function is called on message received.

    :param message: Delivered message.
    :return:
    """
    print("Message:", message.body)

    # Acknowledge that we handled the message without any issues.
    message.ack()

    # Reject the message.
    # message.reject()

    # Reject the message, and put it back in the queue.
    # message.reject(requeue=True)
```

Ensuite, nous devons configurer la connexion au serveur RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

Après cela, nous devons configurer un canal. Chaque connexion peut avoir plusieurs canaux et, en général, lors de l'exécution de tâches multithread, il est recommandé (mais pas obligatoire) d'en avoir une par thread.

```
channel = connection.channel()
```

Une fois notre chaîne configurée, nous devons informer RabbitMQ que nous souhaitons commencer à consommer des messages. Dans ce cas, nous utiliserons notre fonction `on_message` précédemment définie pour gérer tous nos messages consommés.

La file d'attente que nous allons écouter sur le serveur RabbitMQ va être `simple_queue` , et nous disons également à RabbitMQ que nous `simple_queue` réception de tous les messages entrants une fois que nous en aurons fini.

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

Enfin, nous devons démarrer la boucle IO pour commencer le traitement des messages fournis par le serveur RabbitMQ.

```
channel.start_consuming(to_tuple=False)
```

Comment publier des messages sur RabbitMQ

Commencez par importer la bibliothèque.

```
from amqpstorm import Connection
from amqpstorm import Message
```

Ensuite, nous devons ouvrir une connexion au serveur RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

Après cela, nous devons configurer un canal. Chaque connexion peut avoir plusieurs canaux et, en général, lors de l'exécution de tâches multithread, il est recommandé (mais pas obligatoire) d'en avoir une par thread.

```
channel = connection.channel()
```

Une fois notre chaîne configurée, nous pouvons commencer à préparer notre message.

```
# Message Properties.
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
}

# Create the message.
message = Message.create(channel=channel, body='Hello World!', properties=properties)
```

Maintenant, nous pouvons publier le message en appelant simplement `publish` et en fournissant une `routing_key` . Dans ce cas, nous allons envoyer le message à une file d'attente appelée `simple_queue` .

```
message.publish(routing_key='simple_queue')
```

Comment créer une file d'attente différée dans RabbitMQ

Nous devons d'abord configurer deux canaux de base, un pour la file d'attente principale et un pour la file d'attente des délais. Dans mon exemple à la fin, j'inclus quelques indicateurs supplémentaires qui ne sont pas requis, mais rend le code plus fiable; tels que `confirm delivery` , `delivery_mode` et `durable` . Vous pouvez trouver plus d'informations à ce sujet dans le manuel RabbitMQ.

Après avoir configuré les canaux, nous ajoutons une liaison au canal principal que nous pouvons utiliser pour envoyer des messages du canal de délai à notre file d'attente principale.

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

Ensuite, nous devons configurer notre canal de délai pour transmettre les messages à la file d'attente principale une fois qu'ils ont expiré.

```
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000,
    'x-dead-letter-exchange': 'amq.direct',
    'x-dead-letter-routing-key': 'hello'
})
```

- x-message-ttl *(Message - Temps de vie)*

  Ceci est normalement utilisé pour supprimer automatiquement les anciens messages dans la file d'attente après une durée spécifique, mais en ajoutant deux arguments facultatifs, nous pouvons changer ce comportement et déterminer ce paramètre en millisecondes en combien de temps les messages resteront dans la file d'attente.

- x-dead-letter-routing-key

  Cette variable nous permet de transférer le message vers une autre file d'attente une fois qu'ils ont expiré, au lieu du comportement par défaut de la supprimer complètement.

- échange de lettres mortes

  Cette variable détermine quel Exchange est utilisé pour transférer le message de hello_delay vers la file d'attente hello.

Publication dans la file d'attente des délais

Lorsque vous avez terminé de configurer tous les paramètres de base de Pika, vous envoyez simplement un message à la file d'attente de délai en utilisant la publication de base.

```
delay_channel.basic.publish(exchange='',
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mod': 2})
```

Une fois que vous avez exécuté le script, vous devriez voir les files d'attente suivantes créées dans votre module de gestion RabbitMQ.

Overview					Messages				Messa
Name	**Exclusive**	Parameters	**Policy**	**Status**	**Ready**	**Unacked**	**Total**	**incoming**	**deliv**
hello		D		Idle	1	0	1		
hello_delay		TTL DLX DLK D		Idle	0	0	0	0.00/s	

Exemple.

```
from amqpstorm import Connection

connection = Connection('127.0.0.1', 'guest', 'guest')

# Create normal 'Hello World' type channel.
channel = connection.channel()
channel.confirm_deliveries()
channel.queue.declare(queue='hello', durable=True)

# We need to bind this channel to an exchange, that will be used to transfer
# messages from our delay queue.
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')

# Create our delay channel.
delay_channel = connection.channel()
delay_channel.confirm_deliveries()

# This is where we declare the delay, and routing for our delay channel.
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000, # Delay until the message is transferred in milliseconds.
    'x-dead-letter-exchange': 'amq.direct', # Exchange used to transfer the message from A to
B.
    'x-dead-letter-routing-key': 'hello' # Name of the queue we want the message transferred
to.
})

delay_channel.basic.publish(exchange='',
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mode': 2})

print("[x] Sent")
```

Lire Introduction à RabbitMQ en utilisant AMQPStorm en ligne:
https://riptutorial.com/fr/python/topic/3373/introduction-a-rabbitmq-en-utilisant-amqpstorm

Chapitre 92: Iterables et Iterators

Examples

Itérateur vs Iterable vs Générateur

Un **iterable** est un objet qui peut renvoyer un **itérateur** . Tout objet avec un état possédant une méthode `__iter__` et `__iter__` un itérateur est une itération. Il peut également s'agir d'un objet *sans* état qui implémente une méthode `__getitem__` . - La méthode peut prendre des indices (à partir de zéro) et élever une `IndexError` lorsque les indices ne sont plus valides.

La classe `str` de Python est un exemple d'itérable `__getitem__` .

Un **itérateur** est un objet qui produit la valeur suivante dans une séquence lorsque vous appelez `next(*object*)` sur un objet. De plus, tout objet avec une méthode `__next__` est un itérateur. Un itérateur déclenche `StopIteration` après avoir épuisé l'itérateur et *ne peut pas* être réutilisé à ce stade.

Classes itérables:

Les classes itérables définissent une `__iter__` et une méthode `__next__` . Exemple de classe itérable:

```
class MyIterable:

    def __iter__(self):

         return self

    def __next__(self):
         #code

#Classic iterable object in older versions of python, __getitem__ is still supported...
class MySequence:

    def __getitem__(self, index):

         if (condition):
             raise IndexError
         return (item)

 #Can produce a plain `iterator` instance by using iter(MySequence())
```

Essayer d'instancier la classe abstraite du module de `collections` pour mieux voir cela.

Exemple:

Python 2.x 2.3

```
import collections
>>> collections.Iterator()
```

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods next
```

Python 3.x 3.0

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods __next__
```

Gérez la compatibilité de Python 3 pour les classes itérables dans Python 2 en procédant comme suit:

Python 2.x 2.3

```
class MyIterable(object): #or collections.Iterator, which I'd recommend....

     ....

     def __iter__(self):

          return self

     def next(self): #code

     __next__ = next
```

Les deux sont maintenant des itérateurs et peuvent être parcourus en boucle:

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

Les générateurs sont des moyens simples de créer des itérateurs. Un générateur *est* un itérateur et un itérateur est une itération.

Qu'est-ce qui peut être itérable

Iterable peut être tout ce dont les éléments sont reçus *un par un, en avant uniquement* . Les collections Python intégrées sont itérables:

```
[1, 2, 3]     # list, iterate over items
(1, 2, 3)     # tuple
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict, iterate over keys
```

Les générateurs retournent les itérables:

```
def foo():  # foo isn't iterable yet...
    yield 1

res = foo()  # ...but res already is
```

Itérer sur la totalité des itérables

```
s = {1, 2, 3}

# get every element in s
for a in s:
    print a  # prints 1, then 2, then 3

# copy into list
l1 = list(s)  # l1 = [1, 2, 3]

# use list comprehension
l2 = [a * 2 for a in s if a > 2]  # l2 = [6]
```

Vérifier un seul élément dans iterable

Utilisez la décompression pour extraire le premier élément et assurez-vous qu'il est le seul:

```
a, = iterable

def foo():
    yield 1

a, = foo()  # a = 1

nums = [1, 2, 3]
a, = nums  # ValueError: too many values to unpack
```

Extraire des valeurs une par une

Commencez avec `iter()` intégré pour obtenir un **itérateur** sur iterable et utilisez `next()` pour obtenir les éléments un par un jusqu'à `StopIteration` que `StopIteration` soit `StopIteration` pour `StopIteration` la fin:

```
s = {1, 2}   # or list or generator or even iterator
i = iter(s)  # get iterator
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # raises StopIteration
```

Iterator n'est pas réentrant!

```
def gen():
    yield 1

iterable = gen()
for a in iterable:
    print a

# What was the first item of iterable? No way to get it now.
# Only to get a new iterator
gen()
```

Lire Iterables et Iterators en ligne: https://riptutorial.com/fr/python/topic/2343/iterables-et-iterators

Chapitre 93: kivy - Framework Python multiplate-forme pour le développement NUI

Introduction

NUI: Une interface utilisateur naturelle (NUI) est un système d'interaction homme-machine que l'utilisateur utilise à travers des actions intuitives liées au comportement humain naturel au quotidien.

Kivy est une bibliothèque Python pour le développement d'applications multimédias activées par multi-touch pouvant être installées sur différents appareils. Multi-touch fait référence à la capacité d'une surface de détection tactile (généralement un écran tactile ou un trackpad) à détecter ou détecter simultanément l'entrée de deux points de contact ou plus.

Examples

Première App

Pour créer une application kivy

1. sous classe la classe **app**
2. Implémentez la méthode de **construction** , qui renverra le widget.
3. Instancier la classe et invoquer la **course** .

```
from kivy.app import App
from kivy.uix.label import Label

class Test(App):
    def build(self):
        return Label(text='Hello world')

if __name__ == '__main__':
    Test().run()
```

Explication

```
from kivy.app import App
```

L'instruction ci-dessus importera l' **application de** classe parent. Ce sera présent dans votre répertoire d'installation votre_répertoire_installation / kivy / app.py

```
from kivy.uix.label import Label
```

L'instruction ci-dessus importera l' **étiquette de l'** élément ux. Tous les éléments ux sont présents dans votre répertoire d'installation répertoire_installation / kivy / uix /.

```
class Test(App):
```

La déclaration ci-dessus est pour créer votre application et le nom de la classe sera le nom de votre application. Cette classe est héritée de la classe app parent.

```
def build(self):
```

L'instruction ci-dessus remplace la méthode de génération de la classe d'application. Ce qui renverra le widget à afficher lorsque vous lancerez l'application.

```
return Label(text='Hello world')
```

L'instruction ci-dessus est le corps de la méthode de génération. Il retourne l'étiquette avec son texte **Bonjour tout le monde** .

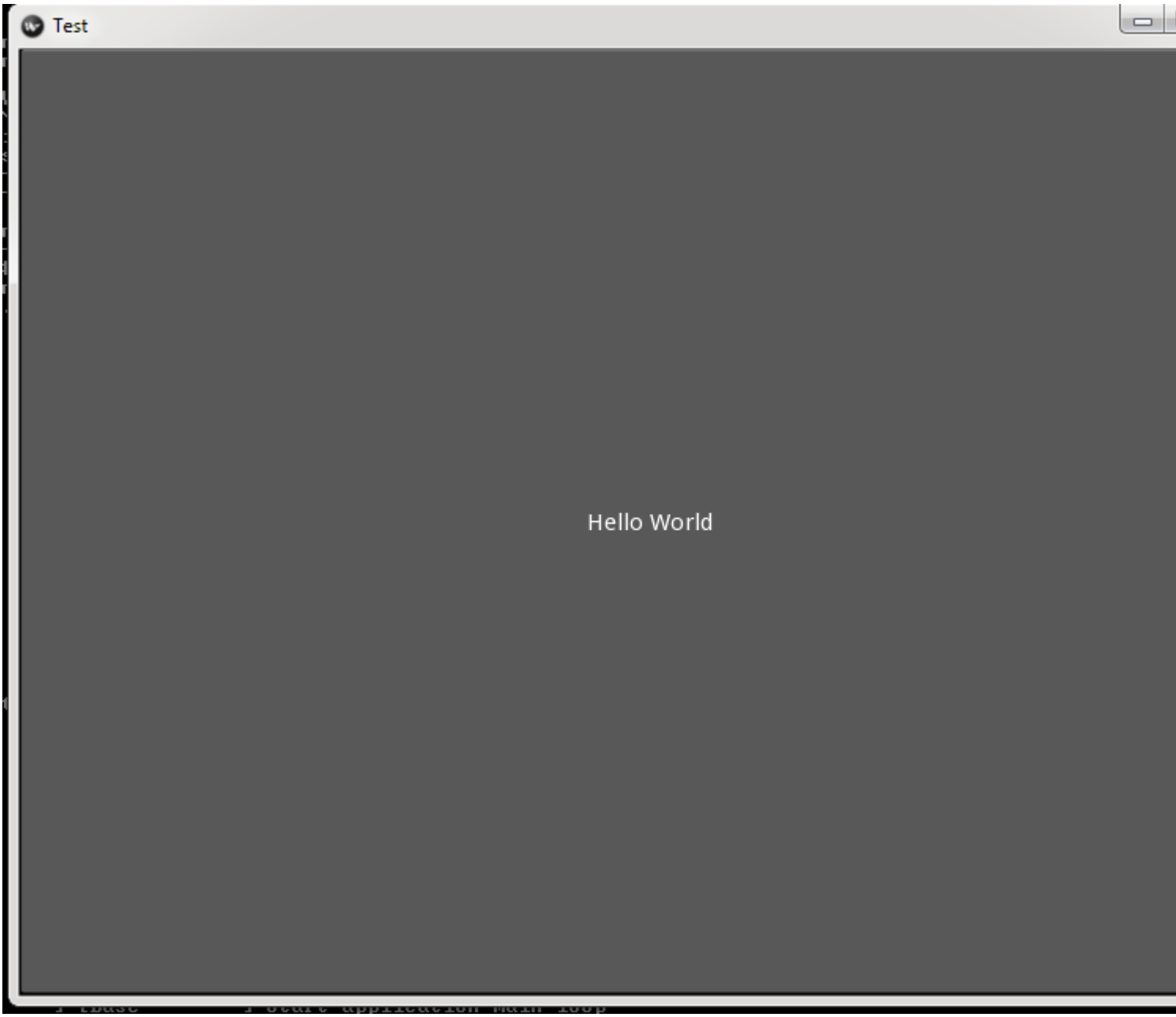
```
if __name__ == '__main__':
```

L'instruction ci-dessus est le point d'entrée d'où l'interpréteur de python commence à exécuter votre application.

```
Test().run()
```

La déclaration ci-dessus Initialise votre classe Test en créant son instance. Et invoquez la fonction de classe d'application run ().

Votre application ressemblera à l'image ci-dessous.



Lire kivy - Framework Python multiplate-forme pour le développement NUI en ligne: https://riptutorial.com/fr/python/topic/10743/kivy---framework-python-multiplate-forme-pour-le-developpement-nui

Chapitre 94: l'audio

Examples

Audio Avec Pyglet

```
import pyglet
audio = pyglet.media.load("audio.wav")
audio.play()
```

Pour plus d'informations, voir pyglet

Travailler avec des fichiers WAV

winsound

- Environnement Windows

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```

vague

- Support mono / stéréo
- Ne supporte pas la compression / décompression

```
import wave
with wave.open("path_to_wav_file.wav", "rb") as wav_file:    # Open WAV file in read-only
mode.
    # Get basic information.
    n_channels = wav_file.getnchannels()      # Number of channels. (1=Mono, 2=Stereo).
    sample_width = wav_file.getsampwidth()    # Sample width in bytes.
    framerate = wav_file.getframerate()       # Frame rate.
    n_frames = wav_file.getnframes()          # Number of frames.
    comp_type = wav_file.getcomptype()        # Compression type (only supports "NONE").
    comp_name = wav_file.getcompname()        # Compression name.

    # Read audio data.
    frames = wav_file.readframes(n_frames)    # Read n_frames new frames.
    assert len(frames) == sample_width * n_frames

# Duplicate to a new WAV file.
with wave.open("path_to_new_wav_file.wav", "wb") as wav_file:    # Open WAV file in write-only
mode.
    # Write audio data.
    params = (n_channels, sample_width, framerate, n_frames, comp_type, comp_name)
    wav_file.setparams(params)
    wav_file.writeframes(frames)
```

Convertir n'importe quel fichier son avec python et ffmpeg

```
from subprocess import check_call

ok = check_call(['ffmpeg','-i','input.mp3','output.wav'])
if ok:
    with open('output.wav', 'rb') as f:
        wav_file = f.read()
```

Remarque:

- http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is--here-even-a-difference
- Quelles sont les différences et les similitudes entre ffmpeg, libav et avconv?

Jouer les bips de Windows

Windows fournit une interface explicite grâce à laquelle le module `winsound` vous permet de jouer des signaux sonores bruts à une fréquence et une durée données.

```
import winsound
freq = 2500 # Set frequency To 2500 Hertz
dur = 1000 # Set duration To 1000 ms == 1 second
winsound.Beep(freq, dur)
```

Lire l'audio en ligne: https://riptutorial.com/fr/python/topic/8189/l-audio

Chapitre 95: L'interpréteur (console de ligne de commande)

Examples

Obtenir de l'aide générale

Si la fonction d' `help` est appelée dans la console sans aucun argument, Python présente une console d'aide interactive où vous pouvez vous renseigner sur les modules Python, les symboles, les mots-clés, etc.

```
>>> help()

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.4/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".
```

Se référant à la dernière expression

Pour obtenir la valeur du dernier résultat de votre dernière expression dans la console, utilisez un trait de soulignement _ .

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

Cette valeur de soulignement magique n'est mise à jour que lorsque vous utilisez une expression python qui génère une valeur. La définition de fonctions ou de boucles ne modifie pas la valeur. Si l'expression déclenche une exception, il n'y aura aucune modification à _ .

```
>>> "Hello, {0}".format("World")
'Hello, World'
>>> _
'Hello, World'
>>> def wontchangethings():
...     pass
```

```
>>> _
'Hello, World'
>>> 27 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> _
'Hello, World'
```

Rappelez-vous que cette variable magique n'est disponible que dans l'interpréteur de python interactif. Les scripts en cours ne le feront pas.

Ouvrir la console Python

La console pour la version principale de Python peut généralement être ouverte en tapant `py` dans votre console Windows ou dans `python` sur d'autres plates-formes.

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Si vous avez plusieurs versions, leurs exécutables seront par défaut mappés respectivement sur `python2` ou `python3` .

Cela dépend bien sûr des exécutables Python présents dans votre PATH.

La variable PYTHONSTARTUP

Vous pouvez définir une variable d'environnement appelée PYTHONSTARTUP pour la console de Python. Chaque fois que vous entrez dans la console Python, ce fichier sera exécuté, vous permettant d'ajouter des fonctionnalités supplémentaires à la console, telles que l'importation automatique de modules couramment utilisés.

Si la variable PYTHONSTARTUP a été définie sur l'emplacement d'un fichier contenant ceci:

```
print("Welcome!")
```

L'ouverture de la console Python entraînerait alors cette sortie supplémentaire:

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Welcome!
>>>
```

Arguments de ligne de commande

Python a une variété de commutateurs de ligne de commande qui peuvent être passés à `py` . Ceux-ci peuvent être trouvés en effectuant `py --help` , qui donne cette sortie sur Python 3.4:

```
Python Launcher

usage: py [ launcher-arguments ] [ python-arguments ] script [ script-arguments ]

Launcher arguments:

-2     : Launch the latest Python 2.x version
-3     : Launch the latest Python 3.x version
-X.Y   : Launch the specified Python version
-X.Y-32: Launch the specified 32bit Python version

The following help text is from Python:

usage: G:\Python34\python.exe [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I     : isolate Python from the user's environment (implies -E and -s)
-m mod : run library module as a script (terminates option list)
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO    : remove doc-strings in addition to the -O optimizations
-q     : don't print version and copyright messages on interactive startup
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-u     : unbuffered binary stdout and stderr, stdin always buffered;
         also PYTHONUNBUFFERED=x
         see man page for details on internal buffering relating to '-u'
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
-W arg : warning control; arg is action:message:category:module:lineno
         also PYTHONWARNINGS=arg
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt : set implementation-specific option
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ';'-separated list of directories prefixed to the
               default module search path.  The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>;<exec_prefix>).
               The default module search path uses <prefix>\lib.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
PYTHONHASHSEED: if this variable is set to 'random', a random value is used
   to seed the hashes of str, bytes and datetime objects.  It can also be
   set to an integer in the range [0,4294967295] to get hash values with a
   predictable seed.
```

Obtenir de l'aide sur un objet

La console Python ajoute une nouvelle fonction, `help` , qui peut être utilisée pour obtenir des informations sur une fonction ou un objet.

Pour une fonction, `help` imprime sa signature (arguments) et sa documentation, si la fonction en a une.

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Pour un objet, `help` répertorie la docstring de l'objet et les différentes fonctions membres de l'objet.

```
>>> x = 2
>>> help(x)
Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value...
```

Lire L'interpréteur (console de ligne de commande) en ligne:
https://riptutorial.com/fr/python/topic/2473/l-interpreteur--console-de-ligne-de-commande-

Chapitre 96: La déclaration de passage

Syntaxe

- passer

Remarques

Pourquoi voudriez-vous jamais dire à l'interprète de ne rien faire explicitement? Python a l'exigence syntaxique que les blocs de code (after `if` , `except` , `def` , `class` etc.) ne peuvent pas être vides.

Mais parfois, un bloc de code vide est utile en soi. Un bloc de `class` vide peut définir une nouvelle classe différente, telle qu'une exception pouvant être interceptée. Un bloc vide `except` peut être le moyen le plus simple d'exprimer «demander pardon plus tard» s'il n'y a rien à demander pour le pardon. Si un itérateur effectue toutes les tâches lourdes, une boucle vide `for` lancer l'itérateur peut être utile.

Par conséquent, si rien n'est supposé se produire dans un bloc de code, un `pass` est nécessaire pour qu'un tel bloc ne produise pas une `IndentationError` d' `IndentationError` . Alternativement, n'importe quelle instruction (incluant juste un terme à évaluer, comme le littéral `Ellipsis` `...` ou une chaîne, le plus souvent une docstring) peut être utilisée, mais le `pass` indique clairement que rien n'est supposé se produire et n'a pas besoin être réellement évalué et (au moins temporairement) stocké en mémoire. Voici un petit recueil annoté des utilisations les plus fréquentes de `pass` qui ont traversé mon chemin - ainsi que quelques commentaires sur les bonnes et mauvaises pratiques.

- Ignorer (tout ou partie) un certain type d' `Exception` (exemple de `xml`):

```
try:
    self.version = "Expat %d.%d.%d" % expat.version_info
except AttributeError:
    pass # unknown
```

Note: Ignorer tous les types de relances, comme dans l'exemple suivant à partir de `pandas` , est généralement considéré comme une mauvaise pratique, car il `SystemExit` également les exceptions qui devraient probablement être transmises à l'appelant, par exemple `KeyboardInterrupt` ou `SystemExit` (ou même `HardwareIsOnFireError` . vous n'exécutez pas une boîte personnalisée avec des erreurs spécifiques définies, que certaines applications appelantes souhaitent connaître?).

```
try:
    os.unlink(filename_larry)
except:
    pass
```

Au lieu d'utiliser au moins `except Error:` ou dans ce cas, `except OSError:` préférence `except OSError:` est considéré comme une pratique bien meilleure. Une analyse rapide de tous les modules python que j'ai installés m'a permis de constater que plus de 10% des modules, à l' `except ...: pass` , interceptaient toutes les exceptions.

- Dériver une classe d'exception qui n'ajoute pas de nouveau comportement (par exemple, dans `scipy`):

```
class CompileError(Exception):
    pass
```

De même, les classes destinées à la classe de base abstraite ont souvent un `__init__` vide explicite ou d'autres méthodes que les sous-classes sont supposées dériver. (p.ex. `pebl`)

```
class _BaseSubmittingController(_BaseController):
    def submit(self, tasks): pass
    def retrieve(self, deferred_results): pass
```

- Tester ce code s'exécute correctement pour quelques valeurs de test, sans se soucier des résultats (de `mpmath`):

```
for x, error in MDNewton(mp, f, (1,-2), verbose=0,
                         norm=lambda x: norm(x, inf)):
    pass
```

- Dans les définitions de classe ou de fonction, un docstring est souvent déjà en place en tant qu'instruction *obligatoire* à exécuter dans le bloc. Dans ce cas, le bloc peut contenir `pass` *en plus* de la docstring pour dire, par exemple dans « Ceci est en effet destiné à ne rien faire. » `pebl` :

```
class ParsingError(Exception):
    """Error encountered while parsing an ill-formed datafile."""
    pass
```

- Dans certains cas, `pass` est utilisé comme un espace réservé pour dire "Cette méthode / class / if-block / ... n'a pas encore été implémentée, mais ce sera l'endroit pour le faire", bien que je préfère personnellement le littéral `Ellipsis` `...` (REMARQUE: python-3 uniquement) afin de faire la distinction entre ceci et le «non-op» intentionnel dans l'exemple précédent. Par exemple, si j'écris un modèle en gros traits, je pourrais écrire

```
def update_agent(agent):
    ...
```

où d'autres pourraient avoir

```
def update_agent(agent):
    pass
```

avant

```
def time_step(agents):
    for agent in agents:
        update_agent(agent)
```

pour vous rappeler de remplir la fonction `update_agent` ultérieurement, mais lancez des tests pour voir si le reste du code se comporte comme prévu. (Une troisième option pour ce cas est `raise NotImplementedError` Ceci est utile notamment pour deux cas. Soit *« Cette méthode abstraite doit être mis en œuvre par chaque sous - classe, il n'y a pas moyen générique de définir dans cette classe de base »* ou *« Cette fonction , avec ce nom, n'est pas encore implémenté dans cette version, mais voici à quoi sa signature ressemblera ”*)

Examples

Ignorer une exception

```
try:
    metadata = metadata['properties']
except KeyError:
    pass
```

Créer une nouvelle exception pouvant être interceptée

```
class CompileError(Exception):
    pass
```

Lire La déclaration de passage en ligne: https://riptutorial.com/fr/python/topic/6891/la-declaration-de-passage

Chapitre 97: La fonction d'impression

Examples

Notions de base sur l'impression

Dans Python 3 et versions ultérieures, `print` est une fonction plutôt qu'un mot clé.

```
print('hello world!')
# out: hello world!

foo = 1
bar = 'bar'
baz = 3.14

print(foo)
# out: 1
print(bar)
# out: bar
print(baz)
# out: 3.14
```

Vous pouvez également transmettre un certain nombre de paramètres à `print` :

```
print(foo, bar, baz)
# out: 1 bar 3.14
```

Une autre façon d' `print` plusieurs paramètres est d'utiliser un `+`

```
print(str(foo) + " " + bar + " " + str(baz))
# out: 1 bar 3.14
```

Ce que vous devez faire attention lorsque vous utilisez `+` pour imprimer plusieurs paramètres, c'est que le type des paramètres doit être identique. Si vous essayez d'imprimer l'exemple ci-dessus sans la conversion en `string` une erreur se produirait, car cela tenterait d'ajouter le nombre `1` à la chaîne `"bar"` et d'ajouter cela au nombre `3.14` .

```
# Wrong:
# type:int  str  float
print(foo + bar + baz)
# will result in an error
```

En effet, le contenu de l' `print` sera évalué en premier:

```
print(4 + 5)
# out: 9
print("4" + "5")
# out: 45
print([4] + [5])
# out: [4, 5]
```

Sinon, l'utilisation d'un `+` peut être très utile pour un utilisateur pour lire la sortie de variables. Dans l'exemple ci-dessous, la sortie est très facile à lire!

Le script ci-dessous illustre cette

```
import random
#telling python to include a function to create random numbers
randnum = random.randint(0, 12)
#make a random number between 0 and 12 and assign it to a variable
print("The randomly generated number was - " + str(randnum))
```

Vous pouvez empêcher l' `print` fonction d'impression automatique en utilisant une nouvelle ligne de la `end` paramètre:

```
print("this has no newline at the end of it... ", end="")
print("see?")
# out: this has no newline at the end of it... see?
```

Si vous souhaitez écrire dans un fichier, vous pouvez le transmettre en tant que `file` paramètres:

```
with open('my_file.txt', 'w+') as my_file:
    print("this goes to the file!", file=my_file)
```

cela va au fichier!

Paramètres d'impression

Vous pouvez faire plus que simplement imprimer du texte. `print` également plusieurs paramètres pour vous aider.

Argument `sep` : place une chaîne entre les arguments.

Devez-vous imprimer une liste de mots séparés par une virgule ou une autre chaîne?

```
>>> print('apples','bannas', 'cherries', sep=', ')
apple, bannas, cherries
>>> print('apple','banna', 'cherries', sep=', ')
apple, banna, cherries
>>>
```

`end` argument: utilisez autre chose qu'une nouvelle ligne à la fin

Sans l'argument de `end` , toutes `print()` fonctions `print()` écrivent une ligne, puis vont au début de la ligne suivante. Vous pouvez le changer pour ne rien faire (utilisez une chaîne vide de ''), ou doublez l'espacement entre les paragraphes en utilisant deux nouvelles lignes.

```
>>> print("<a", end=''); print(" class='jidn'" if 1 else "", end=''); print("/>")
<a class='jidn'/>
>>> print("paragraph1", end="\n\n"); print("paragraph2")
paragraph1
```

```
paragraph2
>>>
```

`file` argument: envoie la sortie à un endroit autre que sys.stdout.

Vous pouvez maintenant envoyer votre texte à stdout, à un fichier ou à StringIO sans vous soucier de ce qui vous a été donné. S'il se répète comme un fichier, il fonctionne comme un fichier.

```
>>> def sendit(out, *values, sep=' ', end='\n'):
...     print(*values, sep=sep, end=end, file=out)
...
>>> sendit(sys.stdout, 'apples', 'bannas', 'cherries', sep='\t')
apples    bannas    cherries
>>> with open("delete-me.txt", "w+") as f:
...    sendit(f, 'apples', 'bannas', 'cherries', sep=' ', end='\n')
...
>>> with open("delete-me.txt", "rt") as f:
...     print(f.read())
...
apples bannas cherries

>>>
```

Il y a un quatrième paramètre de `flush` qui force purger le flux.

Lire La fonction d'impression en ligne: https://riptutorial.com/fr/python/topic/1360/la-fonction-d-impression

Chapitre 98: La variable spéciale __name__

Introduction

La variable spéciale `__name__` est utilisée pour vérifier si un fichier a été importé en tant que module ou non, et pour identifier une fonction, une classe, un objet module par leur attribut `__name__` .

Remarques

La variable spéciale Python `__name__` est définie sur le nom du module contenant. Au niveau supérieur (comme dans l'interpréteur interactif ou dans le fichier principal), il est défini sur `'__main__'` . Cela peut être utilisé pour exécuter un bloc d'instructions si un module est exécuté directement plutôt que d'être importé.

L'attribut spécial associé `obj.__name__` se trouve sur les classes, les modules et fonctions importés *(y compris les méthodes)* et donne le nom de l'objet lorsqu'il est défini.

Examples

__name__ == '__main__'

La variable spéciale `__name__` n'est pas définie par l'utilisateur. Il est principalement utilisé pour vérifier si le module est exécuté ou non, car une `import` été effectuée. Pour éviter que votre module `if __name__ == '__main__'` certaines parties de son code lorsqu'il est importé, vérifiez `if __name__ == '__main__'` .

Soit **module_1.py** une seule ligne de long:

```
import module2.py
```

Et voyons ce qui se passe, en fonction de **module2.py**

Situation 1

module2.py

```
print('hello')
```

Exécuter **module1.py** imprimera `hello`
Exécuter **module2.py** va imprimer `hello`

Situation 2

module2.py

```
if __name__ == '__main__':
    print('hello')
```

Exécuter **module1.py** n'imprimera rien
Exécuter **module2.py** va imprimer `hello`

function_class_or_module .__ nom__

L'attribut spécial `__name__` d'une fonction, classe ou module est une chaîne contenant son nom.

```
import os

class C:
    pass

def f(x):
    x += 2
    return x


print(f)
# <function f at 0x029976B0>
print(f.__name__)
# f

print(C)
# <class '__main__.C'>
print(C.__name__)
# C

print(os)
# <module 'os' from '/spam/eggs/'>
print(os.__name__)
# os
```

L'attribut `__name__` n'est toutefois pas le nom de la variable qui fait référence à la classe, à la méthode ou à la fonction, mais plutôt le nom qui lui est donné lors de la définition.

```
def f():
    pass

print(f.__name__)
# f - as expected

g = f
print(g.__name__)
# f - even though the variable is named g, the function is still named f
```

Cela peut être utilisé, entre autres, pour le débogage:

```
def enter_exit_info(func):
    def wrapper(*arg, **kw):
        print '-- entering', func.__name__
        res = func(*arg, **kw)
        print '-- exiting', func.__name__
```

```
        return res
    return wrapper

@enter_exit_info
def f(x):
    print 'In:', x
    res = x + 2
    print 'Out:', res
    return res

a = f(2)

# Outputs:
#     -- entering f
#     In: 2
#     Out: 4
#     -- exiting f
```

Utiliser dans la journalisation

Lors de la configuration de la fonctionnalité de `logging` intégrée, un modèle courant consiste à créer un enregistreur avec le `__name__` du module actuel:

```
logger = logging.getLogger(__name__)
```

Cela signifie que le nom qualifié complet du module apparaîtra dans les journaux, ce qui facilitera la recherche de l'origine des messages.

Lire La variable spéciale __name__ en ligne: https://riptutorial.com/fr/python/topic/1223/la-variable-speciale---name--

Chapitre 99: Le débogage

Examples

Le débogueur Python: débogage progressif avec _pdb_

La bibliothèque standard Python comprend une bibliothèque de débogage interactive appelée *pdb* . *pdb* possède des capacités étendues, la plus couramment utilisée étant la possibilité de passer à travers un programme.

Pour entrer immédiatement dans le débogage, utilisez:

```
python -m pdb <my_file.py>
```

Cela démarrera le débogueur sur la première ligne du programme.

Généralement, vous voudrez cibler une section spécifique du code pour le débogage. Pour ce faire, nous importons la bibliothèque *pdb* et utilisons *set_trace ()* pour interrompre le flux de cet exemple de code perturbé.

```
import pdb

def divide(a, b):
    pdb.set_trace()
    return a/b
    # What's wrong with this? Hint: 2 != 3

print divide(1, 2)
```

L'exécution de ce programme lancera le débogueur interactif.

```
python foo.py
> ~/scratch/foo.py(5)divide()
-> return a/b
(Pdb)
```

Cette commande est souvent utilisée sur une seule ligne pour pouvoir être commentée avec un seul caractère.

```
 import pdf; pdb.set_trace()
```

À l'invite *(Pdb),* les commandes peuvent être entrées. Ces commandes peuvent être des commandes de débogueur ou python. Pour imprimer des variables que nous pouvons utiliser *p* du débogueur, ou *l'impression* de python.

```
(Pdb) p a
1
(Pdb) print a
```

```
1
```

Pour voir la liste de toutes les variables locales à utiliser

```
locals
```

fonction intégrée

Ce sont de bonnes commandes de débogueur à connaître:

```
b <n> | <f>: set breakpoint at line *n* or function named *f*.
# b 3
# b divide
b: show all breakpoints.
c: continue until the next breakpoint.
s: step through this line (will enter a function).
n: step over this line (jumps over a function).
r: continue until the current function returns.
l: list a window of code around this line.
p <var>: print variable named *var*.
# p x
q: quit debugger.
bt: print the traceback of the current execution call stack
up: move your scope up the function call stack to the caller of the current function
down: Move your scope back down the function call stack one level
step: Run the program until the next line of execution in the program, then return control
back to the debugger
next: run the program until the next line of execution in the current function, then return
control back to the debugger
return: run the program until the current function returns, then return control back to the
debugger
continue: continue running the program until the next breakpoint (or set_trace si called
again)
```

Le débogueur peut également évaluer de manière interactive python:

```
-> return a/b
(Pdb) p a+b
3
(Pdb) [ str(m) for m in [a,b]]
['1', '2']
(Pdb) [ d for d in xrange(5)]
[0, 1, 2, 3, 4]
```

Remarque:

Si l'un de vos noms de variable coïncide avec les commandes du débogueur, utilisez un point d'exclamation ' **!** 'avant le var pour faire explicitement référence à la variable et non à la commande du débogueur. Par exemple, il arrive souvent que vous utilisiez le nom de variable ' **c** ' pour un compteur et que vous souhaitiez l'imprimer dans le débogueur. une simple commande ' **c** ' continuerait l'exécution jusqu'au prochain point d'arrêt. Au lieu de cela, utilisez ' **! C** ' pour imprimer la valeur de la variable comme suit:

```
(Pdb) !c
```

```
4
```

Via IPython et ipdb

Si IPython (ou Jupyter) est installé, le débogueur peut être appelé en utilisant:

```
import ipdb
ipdb.set_trace()
```

Une fois atteint, le code va sortir et imprimer:

```
 /home/usr/ook.py(3)<module>()
      1 import ipdb
      2 ipdb.set_trace()
----> 3 print("Hello world!")

ipdb>
```

Clairement, cela signifie que l'on doit éditer le code. Il y a un moyen plus simple:

```
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
                                     color_scheme='Linux',
                                     call_pdb=1)
```

Cela provoquera l'appel du débogueur s'il existe une exception non interceptée.

Débogueur distant

Quelques fois, vous devez déboguer du code python qui est exécuté par un autre processus et dans ce cas, `rpdb` est utile.

> rpdb est un wrapper autour de pdb qui redirige stdin et stdout vers un gestionnaire de socket. Par défaut, il ouvre le débogueur sur le port 4444

Usage:

```
# In the Python file you want to debug.
import rpdb
rpdb.set_trace()
```

Et puis, vous devez exécuter ce terminal pour vous connecter à ce processus.

```
# Call in a terminal to see the output
$ nc 127.0.0.1 4444
```

Et vous obtiendrez pdb promt

```
> /home/usr/ook.py(3)<module>()
-> print("Hello world!")
```

```
(Pdb)
```

Lire Le débogage en ligne: https://riptutorial.com/fr/python/topic/2077/le-debogage

Chapitre 100: Le module base64

Introduction

L'encodage Base 64 représente un schéma courant pour coder le format binaire en chaîne ASCII à l'aide de radix 64. Le module base64 fait partie de la bibliothèque standard, ce qui signifie qu'il s'installe avec Python. La compréhension des octets et des chaînes est essentielle à ce sujet et peut être consultée ici . Cette rubrique explique comment utiliser les différentes fonctionnalités et bases numériques du module base64.

Syntaxe

- base64.b64encode (s, altchars = Aucun)
- base64.b64decode (s, altchars = Aucun, validez = False)
- base64.standard_b64encode (s)
- base64.standard_b64decode (s)
- base64.urlsafe_b64encode (s)
- base64.urlsafe_b64decode (s)
- base64.b32encode (s)
- base64.b32decode (s)
- base64.b16encode (s)
- base64.b16decode (s)
- base64.a85encode (b, foldspaces = false, wrapcol = 0, pad = false, adobe = false)
- base64.a85decode (b, foldpaces = False, adobe = False, ignorechars = b '\ t \ n \ r \ v')
- base64.b85encode (b, pad = false)
- base64.b85decode (b)

Paramètres

Paramètre	La description
`base64.b64encode(s, altchars=None)`	
s	Un objet de type octets
altchars	Un objet de type 2 octets, d'une longueur de 2+ caractères, pour remplacer les caractères «+» et «=» lors de la création de l'alphabet Base64. Les caractères supplémentaires sont ignorés.
`base64.b64decode(s, altchars=None, validate=False)`	
s	Un objet de type octets
altchars	Un objet de type 2 octets, d'une longueur de 2+

Paramètre	La description
	caractères, pour remplacer les caractères «+» et «=» lors de la création de l'alphabet Base64. Les caractères supplémentaires sont ignorés.
valider	Si valide est True, les caractères qui ne figurent **pas** dans l'alphabet Base64 normal ou l'alphabet alternatif **ne** sont **pas** supprimés avant le contrôle de remplissage.
`base64.standard_b64encode(s)`	
s	Un objet de type octets
`base64.standard_b64decode(s)`	
s	Un objet de type octets
`base64.urlsafe_b64encode(s)`	
s	Un objet de type octets
`base64.urlsafe_b64decode(s)`	
s	Un objet de type octets
`b32encode(s)`	
s	Un objet de type octets
`b32decode(s)`	
s	Un objet de type octets
`base64.b16encode(s)`	
s	Un objet de type octets
`base64.b16decode(s)`	
s	Un objet de type octets
`base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)`	
b	Un objet de type octets
espaces de pliage	Si foldspaces a la valeur True, le caractère 'y' sera utilisé au lieu de 4 espaces consécutifs.
wrapcol	Le nombre de caractères avant une nouvelle ligne (0 n'implique aucune nouvelle ligne)
tampon	Si pad est True, les octets sont remplis à un multiple

Paramètre	La description
	de 4 avant l'encodage
adobe	Si adobe a la valeur True, l'enchaînement encodé avec est encadré avec '<~' et '' ~> 'comme utilisé avec les produits Adobe.
`base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')`	
b	Un objet de type octets
espaces de pliage	Si foldspaces a la valeur True, le caractère 'y' sera utilisé au lieu de 4 espaces consécutifs.
adobe	Si adobe a la valeur True, l'enchaînement encodé avec est encadré avec '<~' et '' ~> 'comme utilisé avec les produits Adobe.
ignorechars	Un objet de type octet à ignorer dans le processus de codage
`base64.b85encode(b, pad=False)`	
b	Un objet de type octets
tampon	Si pad est True, les octets sont remplis à un multiple de 4 avant l'encodage
`base64.b85decode(b)`	
b	Un objet de type octets

Remarques

Jusqu'à la sortie de Python 3.4, les fonctions d'encodage et de décodage en base64 fonctionnaient uniquement avec les types `bytes` ou `bytearray` . Maintenant, ces fonctions acceptent tout objet de type octet .

Examples

Base64 de codage et de décodage

Pour inclure le module base64 dans votre script, vous devez d'abord l'importer:

```
import base64
```

Les fonctions d'encodage et de décodage base64 requièrent toutes deux un objet de type octet .

Pour obtenir notre chaîne en octets, nous devons l'encoder en utilisant la fonction de codage intégrée de Python. Le plus souvent, le `UTF-8` est utilisé, mais une liste complète de ces encodages standard (y compris les langages avec des caractères différents) peut être trouvée ici dans la documentation officielle de Python. Voici un exemple d'encodage d'une chaîne en octets:

```
s = "Hello World!"
b = s.encode("UTF-8")
```

La sortie de la dernière ligne serait:

```
b'Hello World!'
```

Le préfixe `b` est utilisé pour indiquer que la valeur est un objet octets.

Pour encoder ces octets en Base64, nous utilisons la fonction `base64.b64encode()` :

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
print(e)
```

Ce code produirait les informations suivantes:

```
b'SGVsbG8gV29ybGQh'
```

qui est toujours dans l'objet octets. Pour obtenir une chaîne de ces octets, nous pouvons utiliser la méthode `decode()` de Python avec le `UTF-8` :

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
s1 = e.decode("UTF-8")
print(s1)
```

La sortie serait alors:

```
SGVsbG8gV29ybGQh
```

Si nous voulions encoder la chaîne puis décoder, nous pourrions utiliser la méthode `base64.b64decode()` :

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base64 Encode the bytes
e = base64.b64encode(b)
# Decoding the Base64 bytes to string
s1 = e.decode("UTF-8")
# Printing Base64 encoded string
print("Base64 Encoded:", s1)
```

```
# Encoding the Base64 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base64 bytes
d = base64.b64decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Comme vous vous en doutez, la sortie serait la chaîne d'origine:

```
Base64 Encoded: SGVsbG8gV29ybGQh
Hello World!
```

Base32 de codage et de décodage

Le module base64 inclut également des fonctions de codage et de décodage pour Base32. Ces fonctions sont très similaires aux fonctions Base64:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base32 Encode the bytes
e = base64.b32encode(b)
# Decoding the Base32 bytes to string
s1 = e.decode("UTF-8")
# Printing Base32 encoded string
print("Base32 Encoded:", s1)
# Encoding the Base32 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base32 bytes
d = base64.b32decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Cela produirait la sortie suivante:

```
Base32 Encoded: JBSWY3DPEBLW64TMMQQQ====
Hello World!
```

Base de codage et de décodage16

Le module base64 inclut également des fonctions de codage et de décodage pour Base16. La base 16 est généralement appelée **hexadécimale** . Ces fonctions sont très similaires aux fonctions Base64 et Base32:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
```

```
# Base16 Encode the bytes
e = base64.b16encode(b)
# Decoding the Base16 bytes to string
s1 = e.decode("UTF-8")
# Printing Base16 encoded string
print("Base16 Encoded:", s1)
# Encoding the Base16 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base16 bytes
d = base64.b16decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Cela produirait la sortie suivante:

```
Base16 Encoded: 48656C6C6F20576F726C6421
Hello World!
```

Codage et décodage ASCII85

Adobe a créé son propre encodage appelé **ASCII85,** similaire à Base85, mais qui a ses différences. Cet encodage est fréquemment utilisé dans les fichiers Adobe PDF. Ces fonctions ont été publiées en version Python 3.4. Sinon, les fonctions `base64.a85encode()` et `base64.a85encode()` sont similaires aux précédentes:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# ASCII85 Encode the bytes
e = base64.a85encode(b)
# Decoding the ASCII85 bytes to string
s1 = e.decode("UTF-8")
# Printing ASCII85 encoded string
print("ASCII85 Encoded:", s1)
# Encoding the ASCII85 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the ASCII85 bytes
d = base64.a85decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

Cela génère les éléments suivants:

```
ASCII85 Encoded: 87cURD]i,"Ebo80
Hello World!
```

Base de codage et de décodage85

Tout comme les fonctions Base64, Base32 et Base16, les fonctions d'encodage et de décodage `base64.b85encode()` sont `base64.b85encode()` et `base64.b85decode()` :

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base85 Encode the bytes
e = base64.b85encode(b)
# Decoding the Base85 bytes to string
s1 = e.decode("UTF-8")
# Printing Base85 encoded string
print("Base85 Encoded:", s1)
# Encoding the Base85 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base85 bytes
d = base64.b85decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

qui produit ce qui suit:

```
Base85 Encoded: NM&qnZy;B1a%^NF
Hello World!
```

Lire Le module base64 en ligne: https://riptutorial.com/fr/python/topic/8678/le-module-base64

Chapitre 101: Le module dis

Examples

Constantes dans le module dis

```
EXTENDED_ARG = 145 # All opcodes greater than this have 2 operands
HAVE_ARGUMENT = 90 # All opcodes greater than this have at least 1 operands

cmp_op = ('<', '<=', '==', '!=', '>', '>=', 'in', 'not in', 'is', 'is ...
       # A list of comparator id's. The indecies are used as operands in some opcodes

# All opcodes in these lists have the respective types as there operands
hascompare = [107]
hasconst = [100]
hasfree = [135, 136, 137]
hasjabs = [111, 112, 113, 114, 115, 119]
hasjrel = [93, 110, 120, 121, 122, 143]
haslocal = [124, 125, 126]
hasname = [90, 91, 95, 96, 97, 98, 101, 106, 108, 109, 116]

# A map of opcodes to ids
opmap = {'BINARY_ADD': 23, 'BINARY_AND': 64, 'BINARY_DIVIDE': 21, 'BIN...
# A map of ids to opcodes
opname = ['STOP_CODE', 'POP_TOP', 'ROT_TWO', 'ROT_THREE', 'DUP_TOP', '...
```

Qu'est-ce que le bytecode Python?

Python est un interpréteur hybride. Lors de l'exécution d'un programme, il l'assemble d'abord dans un *bytecode* qui peut ensuite être exécuté dans l'interpréteur Python (également appelé *machine virtuelle Python*). Le module `dis` de la bibliothèque standard peut être utilisé pour rendre le bytecode Python lisible par l'homme en désassemblant des classes, des méthodes, des fonctions et des objets de code.

```
>>> def hello():
...     print "Hello, World"
...
>>> dis.dis(hello)
  2           0 LOAD_CONST               1 ('Hello, World')
              3 PRINT_ITEM
              4 PRINT_NEWLINE
              5 LOAD_CONST               0 (None)
              8 RETURN_VALUE
```

L'interpréteur Python est basé sur une pile et utilise un système premier entré, dernier sorti.

Chaque code d'opération (code opération) dans le langage d'assemblage Python (le bytecode) prend un nombre fixe d'éléments de la pile et renvoie un nombre fixe d'éléments à la pile. S'il n'y a pas assez d'éléments dans la pile pour un code opération, l'interpréteur Python se bloque, éventuellement sans message d'erreur.

Démontage des modules

Pour démonter un module Python, il faut d'abord le transformer en fichier `.pyc` (compilé Python). Pour ce faire, lancez

```
python -m compileall <file>.py
```

Ensuite, dans un interprète, exécutez

```
import dis
import marshal
with open("<file>.pyc", "rb") as code_f:
    code_f.read(8) # Magic number and modification time
    code = marshal.load(code_f) # Returns a code object which can be disassembled
    dis.dis(code) # Output the disassembly
```

Cela compilera un module Python et affichera les instructions bytecode avec `dis` . Le module n'est jamais importé, il est donc sûr d'utiliser un code non fiable.

Lire Le module dis en ligne: https://riptutorial.com/fr/python/topic/1763/le-module-dis

Chapitre 102: Le module local

Remarques

Python 2 Docs: [https://docs.python.org/2/library/locale.html#locale.currency][1]

Examples

Mise en forme des devises US Dollars Utilisation du module local

```
import locale

locale.setlocale(locale.LC_ALL, '')
Out[2]: 'English_United States.1252'

locale.currency(762559748.49)
Out[3]: '$762559748.49'

locale.currency(762559748.49, grouping=True)
Out[4]: '$762,559,748.49'
```

Lire Le module local en ligne: https://riptutorial.com/fr/python/topic/1783/le-module-local

Chapitre 103: Le module os

Introduction

Ce module fournit un moyen portable d'utiliser les fonctionnalités dépendantes du système d'exploitation.

Syntaxe

- importer os

Paramètres

Paramètre	Détails
Chemin	Un chemin vers un fichier. Le séparateur de chemin peut être déterminé par `os.path.sep` .
Mode	L'autorisation souhaitée, en octal (par exemple, `0700`)

Examples

Créer un répertoire

```
os.mkdir('newdir')
```

Si vous devez spécifier des autorisations, vous pouvez utiliser l'argument de `mode` facultatif:

```
os.mkdir('newdir', mode=0700)
```

Obtenir le répertoire actuel

Utilisez la fonction `os.getcwd()` :

```
print(os.getcwd())
```

Déterminer le nom du système d'exploitation

Le module `os` fournit une interface pour déterminer le type de système d'exploitation sur lequel le code s'exécute actuellement.

```
os.name
```

Cela peut renvoyer l'un des éléments suivants dans Python 3:

- `posix`
- `nt`
- `ce`
- `java`

Des informations plus détaillées peuvent être extraites de `sys.platform`

Supprimer un répertoire

Supprimez le répertoire sur le `path` :

```
os.rmdir(path)
```

Vous ne devez pas utiliser `os.remove()` pour supprimer un répertoire. Cette fonction est `OSError` aux *fichiers* et son utilisation dans les répertoires entraînera une `OSError`

Suivez un lien symbolique (POSIX)

Parfois, vous devez déterminer la cible d'un lien symbolique. `os.readlink` fera:

```
print(os.readlink(path_to_symlink))
```

Modifier les autorisations sur un fichier

```
os.chmod(path, mode)
```

où `mode` est la permission souhaitée, en octal.

makedirs - création récursive d'annuaire

Étant donné un répertoire local avec le contenu suivant:

```
└── dir1
    ├── subdir1
    └── subdir2
```

Nous voulons créer le même sous-répertoire, subdir2, sous un nouveau répertoire dir2, qui n'existe pas encore.

```
import os

os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

En cours d'exécution cela se traduit par

```
├── dir1
│   ├── subdir1
│   └── subdir2
└── dir2
    ├── subdir1
    └── subdir2
```

dir2 n'est créé que la première fois qu'il est nécessaire, pour la création de subdir1.

Si nous avions utilisé **os.mkdir** , nous aurions eu une exception car dir2 n'aurait pas encore existé.

```
    os.mkdir("./dir2/subdir1")
OSError: [Errno 2] No such file or directory: './dir2/subdir1'
```

os.makedirs ne l'aimera pas si le répertoire cible existe déjà. Si nous le relançons à nouveau:

```
OSError: [Errno 17] File exists: './dir2/subdir1'
```

Cependant, cela pourrait facilement être résolu en interceptant l'exception et en vérifiant que le répertoire a bien été créé.

```
try:
    os.makedirs("./dir2/subdir1")
except OSError:
    if not os.path.isdir("./dir2/subdir1"):
        raise

try:
    os.makedirs("./dir2/subdir2")
except OSError:
    if not os.path.isdir("./dir2/subdir2"):
        raise
```

Lire Le module os en ligne: https://riptutorial.com/fr/python/topic/4127/le-module-os

Chapitre 104: Lecture et écriture CSV

Examples

Ecrire un fichier TSV

Python

```
import csv

with open('/tmp/output.tsv', 'wt') as out_file:
    tsv_writer = csv.writer(out_file, delimiter='\t')
    tsv_writer.writerow(['name', 'field'])
    tsv_writer.writerow(['Dijkstra', 'Computer Science'])
    tsv_writer.writerow(['Shelah', 'Math'])
    tsv_writer.writerow(['Aumann', 'Economic Sciences'])
```

Fichier de sortie

```
$ cat /tmp/output.tsv

name    field
Dijkstra    Computer Science
Shelah    Math
Aumann    Economic Sciences
```

En utilisant des pandas

Ecrivez un fichier CSV à partir d'un `dict` ou d'un `DataFrame` .

```
import pandas as pd

d = {'a': (1, 101), 'b': (2, 202), 'c': (3, 303)}
pd.DataFrame.from_dict(d, orient="index")
df.to_csv("data.csv")
```

Lisez un fichier CSV en tant que `DataFrame` et convertissez-le en `dict` :

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

Lire Lecture et écriture CSV en ligne: https://riptutorial.com/fr/python/topic/2116/lecture-et-ecriture-csv

Chapitre 105: Les fonctions

Introduction

Les fonctions en Python fournissent un code organisé, réutilisable et modulaire pour effectuer un ensemble d'actions spécifiques. Les fonctions simplifient le processus de codage, empêchent la logique redondante et facilitent le suivi du code. Cette rubrique décrit la déclaration et l'utilisation des fonctions dans Python.

Python possède de nombreuses *fonctions intégrées* telles que `print()` , `input()` , `len()` . Outre les fonctions intégrées, vous pouvez également créer vos propres fonctions pour effectuer des tâches plus spécifiques, appelées *fonctions définies par l'utilisateur* .

Syntaxe

- def *nom_fonction (arg1, ... argN, * args, kw1, Kw2 = défaut, ..., ** kwargs): déclarations*
- lambda *arg1, ... argN, * args, kw1, kw2 = par défaut, ..., ** kwargs* : *expression*

Paramètres

Paramètre	Détails
arg1 , ..., *argN*	Arguments réguliers
* *args*	Arguments de position sans nom
kw1 , ..., *kwN*	Arguments uniquement liés aux mots clés
** *kwargs*	Le reste des arguments de mots clés

Remarques

5 choses de base que vous pouvez faire avec les fonctions:

- Assigner des fonctions à des variables

```
def f():
  print(20)
y = f
y()
# Output: 20
```

- Définir des fonctions dans d'autres fonctions (fonctions imbriquées)

```
def f(a, b, y):
```

```
    def inner_add(a, b):      # inner_add is hidden from outer code
        return a + b
    return inner_add(a, b)**y
```

- Les fonctions peuvent renvoyer d'autres fonctions

```
def f(y):
    def nth_power(x):
        return x ** y
    return nth_power    # returns a function

squareOf = f(2)         # function that returns the square of a number
cubeOf = f(3)           # function that returns the cube of a number
squareOf(3)             # Output: 9
cubeOf(2)               # Output: 8
```

- Les fonctions peuvent être transmises comme paramètres à d'autres fonctions

```
def a(x, y):
    print(x, y)
def b(fun, str):        # b has two arguments: a function and a string
    fun('Hello', str)
b(a, 'Sophia')           # Output: Hello Sophia
```

- Les fonctions internes ont accès à la portée englobante (fermeture)

```
def outer_fun(name):
    def inner_fun():     # the variable name is available to the inner function
        return "Hello "+ name + "!"
    return inner_fun
greet = outer_fun("Sophia")
print(greet())            # Output: Hello Sophia!
```

Ressources additionnelles

- Plus d'informations sur les fonctions et les décorateurs: https://www.thecodeship.com/patterns/guide-to-python-function-decorators/

Examples

Définir et appeler des fonctions simples

L'utilisation de l'instruction `def` est le moyen le plus courant de définir une fonction en python. Cette instruction est une *instruction composée d'une clause unique* avec la syntaxe suivante:

```
def function_name(parameters):
    statement(s)
```

`function_name` est appelé *identificateur* de la fonction. Comme une définition de fonction est une instruction exécutable, son exécution *lie* le nom de la fonction à l'objet fonction qui peut être

appelé ultérieurement à l'aide de l'identificateur.

`parameters` est une liste facultative d'identifiants qui sont liés aux valeurs fournies en tant qu'arguments lorsque la fonction est appelée. Une fonction peut avoir un nombre arbitraire d'arguments séparés par des virgules.

`statement(s)` - également appelée *corps* de la *fonction* - est une séquence non vide d'instructions exécutée à chaque appel de la fonction. Cela signifie qu'un corps de fonction ne peut pas être vide, comme n'importe quel *bloc en retrait* .

Voici un exemple de définition de fonction simple dont le but est d'imprimer `Hello` chaque fois qu'elle est appelée:

```
def greet():
    print("Hello")
```

Appelons maintenant la fonction `greet()` définie:

```
greet()
# Out: Hello
```

C'est un autre exemple de définition de fonction qui prend un seul argument et affiche la valeur passée à chaque fois que la fonction est appelée:

```
def greet_two(greeting):
    print(greeting)
```

Après cela, la fonction `greet_two()` doit être appelée avec un argument:

```
greet_two("Howdy")
# Out: Howdy
```

Vous pouvez également donner une valeur par défaut à cet argument de fonction:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Maintenant, vous pouvez appeler la fonction sans donner de valeur:

```
greet_two()
# Out: Howdy
```

Vous remarquerez que contrairement à de nombreux autres langages, vous n'avez pas besoin de déclarer explicitement un type de retour de la fonction. Les fonctions Python peuvent renvoyer des valeurs de tout type via le mot-clé `return` . Une fonction peut retourner n'importe quel nombre de types différents!

```
def many_types(x):
    if x < 0:
```

```
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))

# Output:
0
Hello!
```

Tant que cela est géré correctement par l'appelant, il s'agit d'un code Python parfaitement valide.

Une fonction qui arrive à la fin de l'exécution sans instruction de retour renverra toujours `None` :

```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

Comme mentionné précédemment, une définition de fonction doit avoir un corps de fonction, une séquence d'instructions non vide. Par conséquent, l'instruction `pass` est utilisée comme corps de fonction, ce qui est une opération null - quand il est exécuté, rien ne se produit. Il fait ce que cela signifie, il saute. Il est utile en tant qu'espace réservé lorsqu'une instruction est requise sur le plan syntaxique, mais aucun code ne doit être exécuté.

Renvoyer des valeurs de fonctions

Les fonctions peuvent `return` une valeur que vous pouvez utiliser directement:

```
def give_me_five():
    return 5

print(give_me_five())  # Print the returned value
# Out: 5
```

ou enregistrer la valeur pour une utilisation ultérieure:

```
num = give_me_five()
print(num)             # Print the saved returned value
# Out: 5
```

ou utilisez la valeur pour toutes les opérations:

```
print(give_me_five() + 10)
# Out: 15
```

Si le `return` est rencontré dans la fonction, la fonction sera immédiatement fermée et les opérations suivantes ne seront pas évaluées:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

Vous pouvez également `return` plusieurs valeurs (sous la forme d'un tuple):

```
def give_me_two_fives():
    return 5, 5  # Returns two 5

first, second = give_me_two_fives()
print(first)
# Out: 5
print(second)
# Out: 5
```

Une fonction *sans* instruction `return` renvoie implicitement `None` . De même, une fonction avec une déclaration de `return` , mais aucune valeur de retour ou variable ne renvoie `None` .

Définir une fonction avec des arguments

Les arguments sont définis entre parenthèses après le nom de la fonction:

```
def divide(dividend, divisor):  # The names of the function and its arguments
    # The arguments are available by name in the body of the function
    print(dividend / divisor)
```

Le nom de la fonction et sa liste d'arguments sont appelés la *signature* de la fonction. Chaque argument nommé est effectivement une variable locale de la fonction.

Lors de l'appel de la fonction, donnez des valeurs aux arguments en les listant dans l'ordre

```
divide(10, 2)
# output: 5
```

ou spécifiez-les dans n'importe quel ordre en utilisant les noms de la définition de fonction:

```
divide(divisor=2, dividend=10)
# output: 5
```

Définir une fonction avec des arguments facultatifs

Les arguments facultatifs peuvent être définis en attribuant (en utilisant `=`) une valeur par défaut au nom de l'argument:

```
def make(action='nothing'):
    return action
```

L'appel de cette fonction est possible de 3 manières différentes:

```
make("fun")
# Out: fun

make(action="sleep")
# Out: sleep

# The argument is optional so the function will use the default value if the argument is
# not passed in.
make()
# Out: nothing
```

Attention

Les types Mutable (`list` , `dict` , `set` , etc.) doivent être traités avec soin lorsqu'ils sont donnés en tant qu'attribut **par défaut** . Toute mutation de l'argument par défaut le changera de manière permanente. Voir Définition d'une fonction avec des arguments facultatifs mutables .

Définir une fonction avec plusieurs arguments

On peut donner à une fonction autant d'arguments que l'on veut, les seules règles fixes sont que chaque nom d'argument doit être unique et que les arguments optionnels doivent être après les non-optionnels:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

Lorsque vous appelez la fonction, vous pouvez soit donner chaque mot-clé sans le nom, mais l'ordre compte:

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

Ou combiner en donnant les arguments avec nom et sans. Alors ceux avec nom doivent suivre ceux sans, mais l'ordre de ceux avec nom n'a pas d'importance:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

Définir une fonction avec un nombre arbitraire d'arguments

Nombre arbitraire d'arguments de position:

Définir une fonction capable de prendre un nombre arbitraire d’arguments peut être fait en

préfixant l'un des arguments avec un `*`

```
def func(*args):
    # args will be a tuple containing all values that are passed in
    for i in args:
        print(i)

func(1, 2, 3)  # Calling it with 3 arguments
# Out: 1
#      2
#      3

list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values)  # Calling it with list of values, * expands the list
# Out: 1
#      2
#      3

func()  # Calling it without arguments
# No Output
```

Vous **ne pouvez pas** fournir une valeur par défaut pour `args` , par exemple `func(*args=[1, 2, 3])` générera une erreur de syntaxe (ne compilera même pas).

Vous **ne pouvez pas les** fournir par nom lorsque vous appelez la fonction, par exemple `func(*args=[1, 2, 3]) TypeError` une `TypeError` .

Mais si vous avez déjà vos arguments dans un tableau (ou tout autre `Iterable`), vous **pouvez** appeler votre fonction comme ceci: `func(*my_stuff)` .

Ces arguments (`*args`) sont accessibles par index, par exemple `args[0]` renverra le premier argument

Nombre arbitraire d'arguments de mot clé

Vous pouvez prendre un nombre arbitraire d'arguments avec un nom en définissant un argument dans la définition avec **deux** `*` devant lui:

```
def func(**kwargs):
    # kwargs will be a dictionary containing the names as keys and the values as values
    for name, value in kwargs.items():
        print(name, value)

func(value1=1, value2=2, value3=3)   # Calling it with 3 arguments
# Out: value1 1
#      value2 2
#      value3 3

func()                               # Calling it without arguments
# No Out put

my_dict = {'foo': 1, 'bar': 2}
func(**my_dict)                      # Calling it with a dictionary
# Out: foo 1
```

```
#      bar 2
```

Vous **ne pouvez pas les** fournir **sans** noms, par exemple `func(1, 2, 3)` va générer une `TypeError` .

`kwargs` est un dictionnaire python natif. Par exemple, `args['value1']` donnera la valeur de l'argument `value1` . Assurez-vous de vérifier au préalable qu'il existe un tel argument ou qu'une `KeyError` sera `KeyError` .

Attention

Vous pouvez les mélanger avec d'autres arguments facultatifs et obligatoires, mais l'ordre dans la définition est important.

Les arguments **position / mot-clé** viennent en premier. (Arguments requis).
Puis viennent les arguments **arbitraires** `*arg` . (Optionnel).
Ensuite, les arguments **contenant uniquement des mots clés** viennent ensuite. (Champs obligatoires).
Enfin, le **mot-clé arbitraire** `**kwargs` vient. (Optionnel).

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
     pass
```

- `arg1` doit être donné, sinon une `TypeError` est `TypeError` . Il peut être donné comme positionnel (`func(10)`) ou mot-clé argument (`func(arg1=10)`).
- `kwarg1` doit également être donné, mais il ne peut être fourni qu'en tant que mot-clé: `func(kwarg1=10)` .
- `arg2` et `kwarg2` sont optionnels. Si la valeur doit être modifiée, les mêmes règles que pour `arg1` (positionnelle ou mot clé) et `kwarg1` (uniquement mot clé) s'appliquent.
- `*args` intercepte des paramètres de position supplémentaires. Mais notez que `arg1` et `arg2` doivent être fournis comme arguments positionnels pour passer des arguments à `*args` : `func(1, 1, 1, 1)` .
- `**kwargs` attrape tous les paramètres de mots-clés supplémentaires. Dans ce cas, tout paramètre qui n'est pas `arg1` , `arg2` , `kwarg1` ou `kwarg2` . Par exemple: `func(kwarg3=10)` .
- Dans Python 3, vous pouvez utiliser `*` seul pour indiquer que tous les arguments suivants doivent être spécifiés en tant que mots-clés. Par exemple, la fonction `math.isclose` dans Python 3.5 et `math.isclose` ultérieures est définie à l'aide de `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)` , ce qui signifie que les deux premiers arguments peuvent être fournis Les troisième et quatrième paramètres ne peuvent être fournis que sous forme d'arguments par mots clés.

Python 2.x ne prend pas en charge les paramètres par mot clé uniquement. Ce comportement peut être émulé avec `kwargs` :

```
def func(arg1, arg2=10, **kwargs):
    try:
        kwarg1 = kwargs.pop("kwarg1")
```

```
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # function body ...
```

Remarque sur le nommage

La convention de nommage en option des arguments de position `args` et arguments optionnels mot - clé `kwargs` est juste une convention que vous **pouvez** utiliser tous les noms que vous voulez , **mais** il est utile de suivre la convention pour que les autres savent ce que vous faites, *ou vous - même* si s'il vous plaît faites *plus tard.*

Note sur l'unicité

Toute fonction peut être définie avec **aucun ou un** `*args` et **aucun ou un** `**kwargs` mais pas avec plus d'un de chacun. De plus, `*args` **doit** être le dernier argument positionnel et `**kwargs` doit être le dernier paramètre. Toute tentative d'utiliser plus d'un ou l' autre **se** traduira par une exception d'erreur de syntaxe.

Remarque sur les fonctions d'imbrication avec des arguments facultatifs

Il est possible d'imbriquer ces fonctions et la convention habituelle consiste à supprimer les éléments que le code a déjà traités, **mais** si vous transmettez les paramètres, vous devez passer des arguments positionnels optionnels avec un préfixe `*` et des mots-clés facultatifs args avec un préfixe `**` , sinon args with soit passé comme une liste ou tuple et kwargs comme un seul dictionnaire. par exemple:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)

def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

Définition d'une fonction avec des arguments facultatifs mutables

Il y a un problème lors de l'utilisation d' **arguments facultatifs** avec un **type par défaut mutable** (décrit dans Définition d'une fonction avec des arguments facultatifs), ce qui peut potentiellement conduire à un comportement inattendu.

Explication

Ce problème se pose car les arguments par défaut d'une fonction sont initialisés **une fois** , au moment où la fonction est *définie* , et **non** (comme beaucoup d'autres langages) lorsque la fonction est *appelée* . Les valeurs par défaut sont stockées dans la variable membre `__defaults__` l'objet fonction.

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

Pour les types **immuables** (voir Passage d'argument et mutabilité), ce n'est pas un problème car il n'y a aucun moyen de modifier la variable. il ne peut jamais être réaffecté, laissant la valeur d'origine inchangée. Par conséquent, les valeurs suivantes sont garanties pour avoir la même valeur par défaut. Cependant, pour un type **mutable** , la valeur d'origine peut muter, en faisant des appels à ses différentes fonctions membres. Par conséquent, les appels successifs à la fonction ne sont pas garantis pour avoir la valeur par défaut initiale.

```
def append(elem, to=[]):
    to.append(elem)      # This call to append() mutates the default variable "to"
    return to

append(1)
# Out: [1]

append(2)  # Appends it to the internally stored list
# Out: [1, 2]

append(3, [])  # Using a new created list gives the expected result
# Out: [3]

# Calling it again without argument will append to the internally stored list again
append(4)
# Out: [1, 2, 4]
```

> **Remarque:** certains IDE comme PyCharm émettent un avertissement lorsqu'un type mutable est spécifié comme attribut par défaut.

Solution

Si vous voulez vous assurer que l'argument par défaut est toujours celui que vous spécifiez dans la définition de la fonction, la solution consiste à **toujours** utiliser un type immuable comme argument par défaut.

Un idiome commun pour atteindre cet objectif lorsqu'un type mutable est requis par défaut est d'utiliser `None` (immuable) comme argument par défaut, puis d'affecter la valeur par défaut réelle à la variable d'argument si elle est égale à `None` .

```
def append(elem, to=None):
    if to is None:
        to = []

    to.append(elem)
    return to
```

Fonctions Lambda (Inline / Anonymous)

Le mot clé `lambda` crée une fonction en ligne contenant une seule expression. La valeur de cette expression correspond à ce que la fonction renvoie lorsqu'elle est appelée.

Considérez la fonction:

```
def greeting():
    return "Hello"
```

qui, lorsqu'il est appelé comme:

```
print(greeting())
```

estampes:

```
Hello
```

Cela peut être écrit comme une fonction lambda comme suit:

```
greet_me = lambda: "Hello"
```

> Voir la note au bas de cette section concernant l’affectation des lambdas aux variables. Généralement, ne le faites pas.

Cela crée une fonction en ligne avec le nom `greet_me` qui renvoie `Hello` . Notez que vous n'écrivez pas de `return` lors de la création d'une fonction avec lambda. La valeur après `:` est automatiquement renvoyée.

Une fois assigné à une variable, il peut être utilisé comme une fonction normale:

```
print(greet_me())
```

estampes:

```
Hello
```

`lambda` s peut aussi prendre des arguments:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

renvoie la chaîne:

```
HELLO
```

Ils peuvent aussi prendre un nombre arbitraire d'arguments / arguments, comme les fonctions normales.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

estampes:

```
hello ('world',) {'world': 'world'}
```

`lambda` s sont couramment utilisés pour les fonctions courtes qui sont pratiques à définir au point où elles sont appelées (généralement avec `sorted` , `filter` et `map`).

Par exemple, cette ligne trie une liste de chaînes en ignorant leur cas et en ignorant les espaces au début et à la fin:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip().upper())
# Out:
# ['    bAR', 'BaZ    ', ' foo ']
```

Liste de tri en ignorant les espaces blancs:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip())
# Out:
# ['BaZ    ', '    bAR', ' foo ']
```

Exemples avec `map` :

```
sorted( map( lambda s: s.strip().upper(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BAR', 'BAZ', 'FOO']

sorted( map( lambda s: s.strip(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BaZ', 'bAR', 'foo']
```

Exemples avec des listes numériques:

```
my_list = [3, -4, -2, 5, 1, 7]
sorted( my_list, key=lambda x: abs(x))
# Out:
# [1, -2, 3, -4, 5, 7]

list( filter( lambda x: x>0, my_list))
# Out:
# [3, 5, 1, 7]
```

```
list( map( lambda x: abs(x), my_list))
# Out:
[3, 4, 2, 5, 1, 7]
```

On peut appeler d'autres fonctions (avec / sans arguments) depuis une fonction lambda.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

estampes:

```
hello world
```

Ceci est utile car `lambda` peut contenir une seule expression et en utilisant une fonction subsidiaire, on peut exécuter plusieurs instructions.

REMARQUE

Gardez à l'esprit que PEP-8 (le guide de style officiel Python) ne recommande pas d'attribuer les lambdas aux variables (comme nous l'avons fait dans les deux premiers exemples):

> Utilisez toujours une instruction def au lieu d'une instruction d'affectation qui lie directement une expression lambda à un identificateur.
>
> Oui:
>
> ```
> def f(x): return 2*x
> ```
>
> Non:
>
> ```
> f = lambda x: 2*x
> ```
>
> La première forme signifie que le nom de l'objet fonction résultant est spécifiquement `f` au lieu du nom générique `<lambda>` . Ceci est plus utile pour les traces et les représentations de chaînes en général. L'utilisation de l'instruction d'affectation élimine le seul avantage qu'une expression lambda peut offrir sur une instruction `def` explicite (c'est-à-dire qu'elle peut être incorporée dans une expression plus grande).

Argument passant et mutabilité

D'abord, une terminologie:

- **argument (paramètre *actuel*):** la variable réelle transmise à une fonction;
- **paramètre (paramètre *formel*):** la variable de réception utilisée dans une fonction.

En Python, les arguments sont passés par *affectation* (contrairement aux autres langages, où les arguments peuvent être passés par valeur / référence / pointeur).

- La mutation d'un paramètre mutera l'argument (si le type de l'argument est modifiable).

```
def foo(x):         # here x is the parameter
    x[0] = 9        # This mutates the list labelled by both x and y
    print(x)

y = [4, 5, 6]
foo(y)              # call foo with y as argument
# Out: [9, 5, 6]   # list labelled by x has been mutated
print(y)
# Out: [9, 5, 6]   # list labelled by y has been mutated too
```

- Réaffecter le paramètre ne réassigne pas l'argument.

```
def foo(x):         # here x is the parameter, when we call foo(y) we assign y to x
    x[0] = 9        # This mutates the list labelled by both x and y
    x = [1, 2, 3]  # x is now labeling a different list (y is unaffected)
    x[2] = 8        # This mutates x's list, not y's list

y = [4, 5, 6]       # y is the argument, x is the parameter
foo(y)              # Pretend that we wrote "x = y", then go to line 1
y
# Out: [9, 5, 6]
```

En Python, nous n'attribuons pas vraiment de valeurs aux variables, mais nous *lions* (c'est-à-dire affectons, attachons) des variables (considérées comme des *noms*) à des objets.

- **Immuable:** nombres entiers, chaînes, tuples, etc. Toutes les opérations font des copies.
- **Mutable:** listes, dictionnaires, ensembles, etc. Les opérations peuvent ou non muter.

```
x = [3, 1, 9]
y = x
x.append(5)     # Mutates the list labelled by x and y, both x and y are bound to [3, 1, 9]
x.sort()        # Mutates the list labelled by x and y (in-place sorting)
x = x + [4]     # Does not mutate the list (makes a copy for x only, not y)
z = x           # z is x ([1, 3, 9, 4])
x += [6]        # Mutates the list labelled by both x and z (uses the extend function).
x = sorted(x)   # Does not mutate the list (makes a copy for x only).
x
# Out: [1, 3, 4, 5, 6, 9]
y
# Out: [1, 3, 5, 9]
z
# Out: [1, 3, 5, 9, 4, 6]
```

Fermeture

Les fermetures en Python sont créées par des appels de fonction. Ici, l'appel à `makeInc` crée une liaison pour `x` référencée à l'intérieur de la fonction `inc` . Chaque appel à `makeInc` crée une nouvelle instance de cette fonction, mais chaque instance a un lien vers une liaison différente de `x` .

```
def makeInc(x):
  def inc(y):
     # x is "attached" in the definition of inc
     return y + x

  return inc

incOne = makeInc(1)
incFive = makeInc(5)

incOne(5) # returns 6
incFive(5) # returns 10
```

Notez que bien que dans une fermeture régulière la fonction incluse hérite complètement de toutes les variables de son environnement englobant, dans cette construction, la fonction jointe n'a qu'un accès en lecture aux variables héritées mais ne peut pas leur attribuer

```
def makeInc(x):
  def inc(y):
     # incrementing x is not allowed
     x += y
     return x

  return inc

incOne = makeInc(1)
incOne(5) # UnboundLocalError: local variable 'x' referenced before assignment
```

Python 3 offre l'instruction `nonlocal` (variables `nonlocal`) pour réaliser une fermeture complète avec des fonctions imbriquées.

Python 3.x 3.0

```
def makeInc(x):
  def inc(y):
     nonlocal x
     # now assigning a value to x is allowed
     x += y
     return x

  return inc

incOne = makeInc(1)
incOne(5) # returns 6
```

Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle-même dans sa définition. Par exemple, la fonction mathématique, factorielle, définie par `factorial(n) = n*(n-1)*(n-2)*...*3*2*1` . peut être programmé comme

```
def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
```

```
    else:
        return n*factorial(n-1)
```

les sorties sont les suivantes:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

comme prévu. Notez que cette fonction est récursive car le second `return factorial(n-1)` , où la fonction s'appelle elle-même dans sa définition.

Certaines fonctions récursives peuvent être implémentées en utilisant lambda , la fonction factorielle utilisant lambda serait quelque chose comme ceci:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

La fonction produit le même que ci-dessus.

Limite de récursivité

Il y a une limite à la profondeur de la récursion possible, qui dépend de l'implémentation de Python. Lorsque la limite est atteinte, une exception RuntimeError est déclenchée:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))

cursing(0)
# Out: I recursed 1083 times!
```

Il est possible de modifier la limite de profondeur de la récursivité en utilisant `sys.setrecursionlimit(limit)` et en vérifiant cette limite avec `sys.getrecursionlimit()` .

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

À partir de Python 3.5, l'exception est une `RecursionError` , dérivée de `RuntimeError` .

Fonctions imbriquées

Les fonctions en python sont des objets de première classe. Ils peuvent être définis dans n'importe quel domaine

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
    return a
```

Les fonctions qui capturent leur portée peuvent être transmises comme n'importe quel autre objet

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Out: 15
add6(10)
#Out: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

Débarbouillable et dictionnaire

Les fonctions vous permettent de spécifier ces types de paramètres: positionnel, nommé, variable de position, mot-clé args (kwargs). Voici une utilisation claire et concise de chaque type.

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
    print(a, b, c, d, args, kwargs)

>>> unpacking(1, 2)
1 2 45 60 () {}
>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}


>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> args_list = [3]
>>> unpacking(1, 2, *args_list, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, *args_list, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'


>>> pair = (3, 4)
>>> unpacking(1, 2, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *pair)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *pair, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'



>>> args_list = [3, 4]
>>> unpacking(1, 2, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *args_list)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *args_list, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'


>>> arg_dict = {'c':3, 'd':4}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'d':4, 'c':3}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'c':3, 'd':4, 'not_a_parameter': 75}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {'not_a_parameter': 75}
```

```
>>> unpacking(1, 2, *pair, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, 3, 4, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

# Positional arguments take priority over any other form of argument passing
>>> unpacking(1, 2, **arg_dict, c=3)
1 2 3 4 () {'not_a_parameter': 75}
>>> unpacking(1, 2, 3, **arg_dict, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
```

Forcer l'utilisation de paramètres nommés

Tous les paramètres spécifiés après le premier astérisque dans la signature de la fonction sont uniquement des mots clés.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

Dans Python 3, il est possible de placer un seul astérisque dans la signature de la fonction pour s'assurer que les arguments restants ne peuvent être transmis qu'avec des arguments de mots clés.

```
def f(a, b, *, c):
    pass

f(1, 2, 3)
# TypeError: f() takes 2 positional arguments but 3 were given
f(1, 2, c=3)
# No error
```

Lambda récursif utilisant une variable affectée

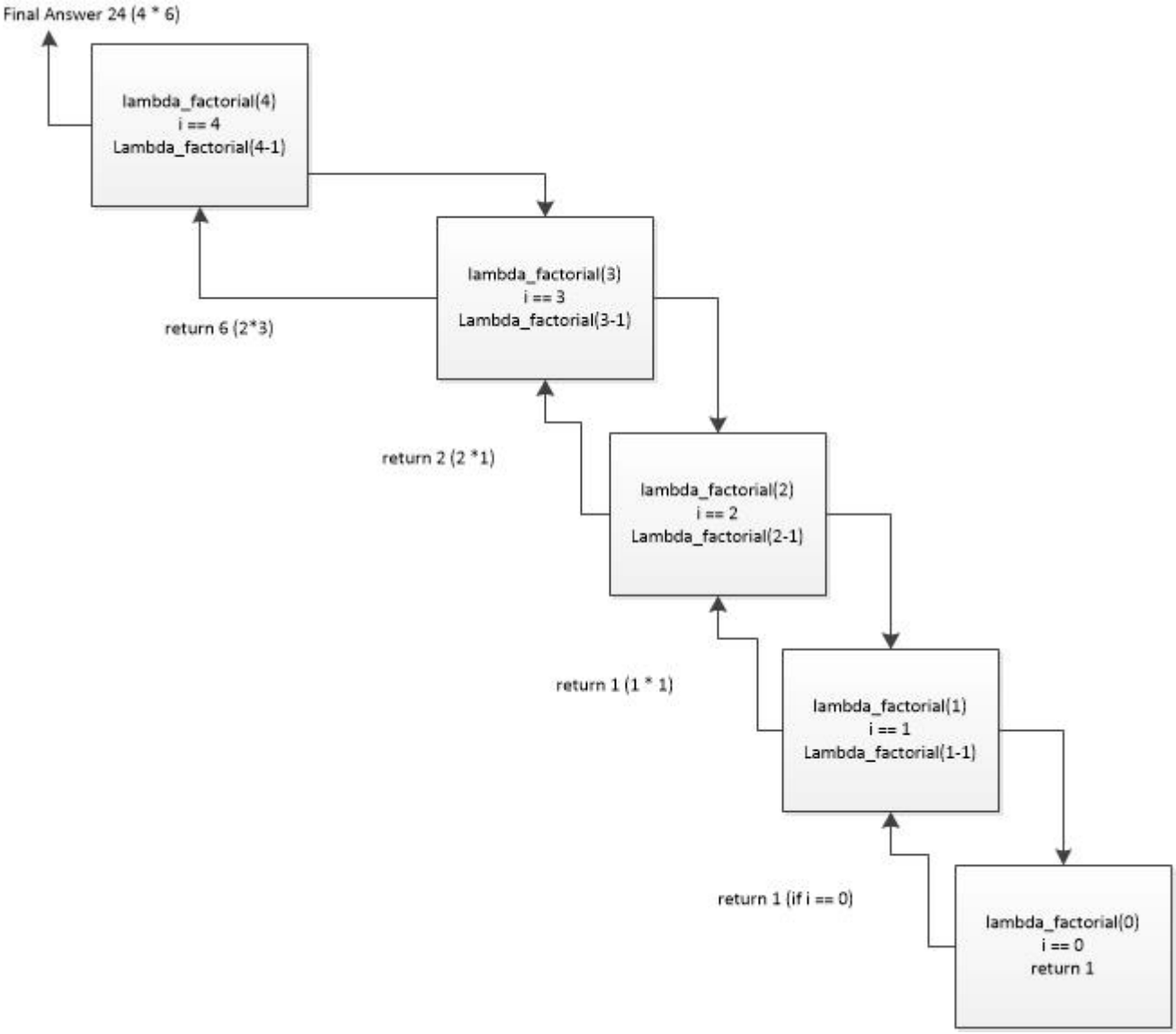
Une méthode pour créer des fonctions lambda récursives consiste à affecter la fonction à une variable, puis à référencer cette variable dans la fonction elle-même. Un exemple courant de ceci est le calcul récursif de la factorielle d'un nombre - tel que montré dans le code suivant:

```
lambda_factorial = lambda i:1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24
```

Description du code

La fonction lambda, à travers son affectation de variable, reçoit une valeur (4) qu'elle évalue et

retourne 1 si elle est 0 sinon elle renvoie la valeur courante (`i`) * un autre calcul par la fonction lambda de la valeur - 1 (`i-1`). Cela continue jusqu'à ce que la valeur passée soit décrémentée à 0 (`return 1`). Un processus qui peut être visualisé comme:



Lire Les fonctions en ligne: https://riptutorial.com/fr/python/topic/228/les-fonctions

Chapitre 106: liste

Introduction

La **liste** Python est une structure de données générale largement utilisée dans les programmes Python. Ils se trouvent dans d'autres langues, souvent appelées *tableaux dynamiques* . Ils sont tous deux *mutables* et un type de données de *séquence* leur permet d'être *indexés* et *découpés* . La liste peut contenir différents types d'objets, y compris d'autres objets de liste.

Syntaxe

- [valeur, valeur, ...]
- liste ([itérable])

Remarques

`list` est un type d'itération particulier, mais ce n'est pas le seul qui existe en Python. Parfois, il vaudra mieux utiliser `set` , `tuple` ou `dictionary`

`list` est le nom donné en Python aux tableaux dynamiques (similaire au `vector<void*>` de C ++ ou à `ArrayList<Object>`). Ce n'est pas une liste liée.

L'accès aux éléments se fait en temps constant et est très rapide. L'ajout d'éléments à la fin de la liste est un temps constant amorti, mais de temps en temps, cela peut impliquer une allocation et une copie de la `list` .

Les compréhensions de liste sont liées aux listes.

Examples

Accéder aux valeurs de la liste

Les listes Python sont indexées à zéro et agissent comme des tableaux dans d'autres langages.

```
lst = [1, 2, 3, 4]
lst[0]  # 1
lst[1]  # 2
```

Tenter d'accéder à un index en dehors des limites de la liste `IndexError` une `IndexError` .

```
lst[4]  # IndexError: list index out of range
```

Les indices négatifs sont interprétés comme comptant à partir de la *fin* de la liste.

```
lst[-1]  # 4
```

```
lst[-2]  # 3
lst[-5]  # IndexError: list index out of range
```

Ceci est fonctionnellement équivalent à

```
lst[len(lst)-1]  # 4
```

Les listes permettent d'utiliser la *notation de tranche* comme `lst[start:end:step]` . Le résultat de la notation de tranche est une nouvelle liste contenant des éléments du `start` index à la `end-1` . Si les options sont omis `start` par défaut au début de la liste, `end` à la fin de la liste et l' `step` 1:

```
lst[1:]      # [2, 3, 4]
lst[:3]      # [1, 2, 3]
lst[::2]     # [1, 3]
lst[::-1]    # [4, 3, 2, 1]
lst[-1:0:-1] # [4, 3, 2]
lst[5:8]     # [] since starting index is greater than length of lst, returns empty list
lst[1:10]    # [2, 3, 4] same as omitting ending index
```

Dans cette optique, vous pouvez imprimer une version inversée de la liste en appelant

```
lst[::-1]    # [4, 3, 2, 1]
```

Lorsque vous utilisez des longueurs d'étape de montants négatifs, l'index de départ doit être supérieur à l'index de fin, sinon le résultat sera une liste vide.

```
lst[3:1:-1] # [4, 3]
```

L'utilisation d'indices d'étape négatifs équivaut au code suivant:

```
reversed(lst)[0:2] # 0 = 1 -1
                   # 2 = 3 -1
```

Les indices utilisés sont inférieurs de 1 à ceux utilisés en indexation négative et sont inversés.

Tranchage avancé

Lorsque des listes sont découpées, la `__getitem__()` de l'objet liste est appelée, avec un objet `slice` . Python a une méthode de tranche intégrée pour générer des objets de tranche. Nous pouvons l'utiliser pour *stocker* une tranche et la réutiliser plus tard comme ça,

```
data = 'chandan purohit    22 2000'  #assuming data fields of fixed length
name_slice = slice(0,19)
age_slice = slice(19,21)
salary_slice = slice(22,None)

#now we can have more readable slices
print(data[name_slice]) #chandan purohit
print(data[age_slice]) #'22'
print(data[salary_slice]) #'2000'
```

Cela peut être très utile en fournissant des fonctionnalités de découpage à nos objets en `__getitem__` dans notre classe.

Méthodes de liste et opérateurs pris en charge

En commençant par une liste donnée `a` :

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` - ajoute un nouvel élément à la fin de la liste.

```
# Append values 6, 7, and 7 to the list
a.append(6)
a.append(7)
a.append(7)
# a: [1, 2, 3, 4, 5, 6, 7, 7]

# Append another list
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]

# Append an element of a different type, as list elements do not need to have the same
type
my_string = "hello world"
a.append(my_string)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
```

Notez que la méthode `append()` ajoute uniquement un nouvel élément à la fin de la liste. Si vous ajoutez une liste à une autre liste, la liste que vous ajoutez devient un élément unique à la fin de la première liste.

```
# Appending a list to another list
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
a[8]
# Returns: [8,9]
```

2. `extend(enumerable)` - étend la liste en ajoutant des éléments d'un autre énumérable.

```
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9, 10]

# Extend list by appending all elements from b
a.extend(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]

# Extend list with elements from a non-list enumerable:
a.extend(range(3))
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
```

Les listes peuvent également être concaténées avec l'opérateur `+` . Notez que cela ne

modifie aucune des listes d'origine:

```
a = [1, 2, 3, 4, 5, 6] + [7, 7] + b
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

3. `index(value, [startIndex])` - obtient l'index de la première occurrence de la valeur d'entrée. Si la valeur d'entrée n'est pas dans la liste, une exception `ValueError` est `ValueError` . Si un second argument est fourni, la recherche est lancée à cet index spécifié.

```
a.index(7)
# Returns: 6

a.index(49) # ValueError, because 49 is not in a.

a.index(7, 7)
# Returns: 7

a.index(7, 8) # ValueError, because there is no 7 starting at index 8
```

4. `insert(index, value)` - Insère une `value` juste avant l' `index` spécifié. Ainsi, après l'insertion, le nouvel élément occupe l' `index` position.

```
a.insert(0, 0)  # insert 0 at position 0
a.insert(2, 5)  # insert 5 at position 2
# a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

5. `pop([index])` - supprime et retourne l'élément à l' `index` . Sans argument, il supprime et retourne le dernier élément de la liste.

```
a.pop(2)
# Returns: 5
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
a.pop(8)
# Returns: 7
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# With no argument:
a.pop()
# Returns: 10
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. `remove(value)` - supprime la première occurrence de la valeur spécifiée. Si la valeur fournie est introuvable, une valeur `ValueError` est `ValueError` .

```
a.remove(0)
a.remove(9)
# a: [1, 2, 3, 4, 5, 6, 7, 8]
a.remove(10)
# ValueError, because 10 is not in a
```

7. `reverse()` - inverse la liste sur place et renvoie `None` .

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

Il existe également d' autres moyens d'inverser une liste .

8. `count(value)` - compte le nombre d'occurrences d'une certaine valeur dans la liste.

```
a.count(7)
# Returns: 2
```

9. `sort()` - trie la liste en ordre numérique et lexicographique et retourne `None` .

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

Les listes peuvent également être inversées lorsqu'elles sont triées à l'aide de l'indicateur `reverse=True` dans la méthode `sort()` .

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

Si vous voulez trier les attributs d'objets, vous pouvez utiliser la `key` argument de mot clé:

```
import datetime

class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height

    def __repr__(self):
        return self.name

l = [Person("John Cena", datetime.date(1992, 9, 12), 175),
     Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
     Person("Jon Skeet", datetime.date(1991, 7, 6), 185)]

l.sort(key=lambda item: item.name)
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item.birthday)
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item.height)
# l: [John Cena, Chuck Norris, Jon Skeet]
```

En cas de liste de dicts, le concept est le même:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'height': 175},
```

```
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'height': 180},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'height': 185}]

l.sort(key=lambda item: item['name'])
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item['birthday'])
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Trier par sous dict:

```
import datetime

l = [{'name':'John Cena', 'birthday': datetime.date(1992, 9, 12),'size': {'height': 175,
'weight': 100}},
 {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28),'size' : {'height': 180,
'weight': 90}},
 {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'size': {'height': 185,
'weight': 110}}]

l.sort(key=lambda item: item['size']['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Meilleure façon de trier avec `attrgetter` et `itemgetter`

Les listes peuvent également être triées à l'aide des fonctions `attrgetter` et `itemgetter` du module opérateur. Ceux-ci peuvent aider à améliorer la lisibilité et la réutilisation. Voici quelques exemples,

```
from operator import itemgetter,attrgetter

people = [{'name':'chandan','age':20,'salary':2000},
          {'name':'chetan','age':18,'salary':5000},
          {'name':'guru','age':30,'salary':3000}]
by_age = itemgetter('age')
by_salary = itemgetter('salary')

people.sort(key=by_age) #in-place sorting by age
people.sort(key=by_salary) #in-place sorting by salary
```

`itemgetter` peut également recevoir un index. Ceci est utile si vous voulez trier en fonction des index d'un tuple.

```
list_of_tuples = [(1,2), (3,4), (5,0)]
list_of_tuples.sort(key=itemgetter(1))
print(list_of_tuples) #[(5, 0), (1, 2), (3, 4)]
```

Utilisez l' `attrgetter` si vous souhaitez trier par attributs d'un objet,

```
persons = [Person("John Cena", datetime.date(1992, 9, 12), 175),
           Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
```

```
          Person("Jon Skeet", datetime.date(1991, 7, 6), 185)] #reusing Person class from
above example

person.sort(key=attrgetter('name')) #sort by name
by_birthday = attrgetter('birthday')
person.sort(key=by_birthday) #sort by birthday
```

10. `clear()` - supprime tous les éléments de la liste

    ```
    a.clear()
    # a = []
    ```

11. **Réplication** - la multiplication d'une liste existante par un nombre entier produira une liste plus grande comprenant autant de copies de l'original. Cela peut être utile par exemple pour l'initialisation de la liste:

    ```
    b = ["blah"] * 3
    # b = ["blah", "blah", "blah"]
    b = [1, 3, 5] * 5
    # [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
    ```

    Faites cela si votre liste contient des références à des objets (par exemple une liste de listes), voir Pièges courants - Multiplication des listes et références communes .

12. **Suppression d'éléments** - il est possible de supprimer plusieurs éléments de la liste en utilisant la notation `del` mot-clé et tranche:

    ```
    a = list(range(10))
    del a[::2]
    # a = [1, 3, 5, 7, 9]
    del a[-1]
    # a = [1, 3, 5, 7]
    del a[:]
    # a = []
    ```

13. **Copier**

    L'attribution par défaut "=" assigne une référence de la liste d'origine au nouveau nom. C'est-à-dire que le nom d'origine et le nouveau nom pointent tous deux vers le même objet de liste. Les modifications apportées par l'un d'entre eux seront reflétées dans une autre. Ce n'est souvent pas ce que vous vouliez.

    ```
    b = a
    a.append(6)
    # b: [1, 2, 3, 4, 5, 6]
    ```

    Si vous souhaitez créer une copie de la liste, vous disposez des options ci-dessous.

    Vous pouvez le couper en tranches:

    ```
    new_list = old_list[:]
    ```

Vous pouvez utiliser la fonction list () intégrée:

```
new_list = list(old_list)
```

Vous pouvez utiliser copy.copy générique ():

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

C'est un peu plus lent que list () car il doit d'abord trouver le type de données de old_list.

Si la liste contient des objets et que vous souhaitez les copier également, utilisez le fichier copy.deepcopy () générique:

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Evidemment la méthode la plus lente et la plus gourmande en mémoire, mais parfois inévitable.

Python 3.x 3.0

`copy()` - Retourne une copie peu profonde de la liste

```
    aa = a.copy()
    # aa = [1, 2, 3, 4, 5]
```

Longueur d'une liste

Utilisez `len()` pour obtenir la longueur unidimensionnelle d'une liste.

```
len(['one', 'two'])  # returns 2

len(['one', [2, 3], 'four'])  # returns 3, not 4
```

`len()` fonctionne également sur les chaînes, les dictionnaires et autres structures de données similaires aux listes.

Notez que `len()` est une fonction intégrée et non une méthode d'objet de liste.

Notez également que le coût de `len()` est `O(1)` , ce qui signifie qu'il faudra le même temps pour obtenir la longueur d'une liste, quelle que soit sa longueur.

Itérer sur une liste

Python prend en charge l'utilisation d'une boucle `for` directement sur une liste:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
```

```
    print(item)

# Output: foo
# Output: bar
# Output: baz
```

Vous pouvez également obtenir la position de chaque élément en même temps:

```
for (index, item) in enumerate(my_list):
    print('The item in position {} is: {}'.format(index, item))

# Output: The item in position 0 is: foo
# Output: The item in position 1 is: bar
# Output: The item in position 2 is: baz
```

L'autre façon d'itérer une liste en fonction de la valeur d'index:

```
for i in range(0,len(my_list)):
    print(my_list[i])
#output:
>>>
foo
bar
baz
```

Notez que la modification d'éléments dans une liste lors d'une itération peut avoir des résultats inattendus:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)

# Output: foo
# Output: baz
```

Dans ce dernier exemple, nous avons supprimé le premier élément à la première itération, mais la `bar` été ignorée.

Vérifier si un article est dans une liste

Python facilite la vérification de la présence d'un élément dans une liste. Il suffit d' utiliser le `in` l' opérateur.

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True

'toast' in lst
# Out: False
```

Note: le `in` l' opérateur sur des ensembles est asymptotiquement plus rapide que sur

les listes. Si vous devez l'utiliser à plusieurs reprises sur des listes potentiellement volumineuses, vous pouvez convertir votre `list` en un `set` et tester la présence d'éléments sur l' `set` .

```
slst = set(lst)
'test' in slst
# Out: True
```

Inverser les éléments de la liste

Vous pouvez utiliser la fonction `reversed` qui renvoie un itérateur à la liste inversée:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Notez que la liste "numéros" reste inchangée par cette opération, et reste dans le même ordre qu'elle était à l'origine.

Pour inverser la position, vous pouvez également utiliser la méthode `reverse` .

Vous pouvez également inverser une liste (en réalité, obtenir une copie, la liste d'origine n'est pas affectée) en utilisant la syntaxe de découpage, en définissant le troisième argument (l'étape) comme étant égal à -1:

```
In [1]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: numbers[::-1]
Out[2]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Vérification si la liste est vide

La vacuité d'une liste est associée à la valeur booléenne `False` , vous n'avez donc pas à vérifier `len(lst) == 0` , mais simplement `lst` ou `not lst`

```
lst = []
if not lst:
    print("list is empty")

# Output: list is empty
```

Concaténer et fusionner les listes

1. **La façon** la plus **simple à concaténer** `list1` et `list2` :

   ```
   merged = list1 + list2
   ```

2. **`zip` renvoie une liste de tuples** , où le i-ème tuple contient le i-ème élément de chacune

des séquences d'arguments ou des itérables:

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3']

for a, b in zip(alist, blist):
    print(a, b)

# Output:
# a1 b1
# a2 b2
# a3 b3
```

Si les listes ont des longueurs différentes, le résultat n'inclura que le plus petit nombre d'éléments:

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3', 'b4']
for a, b in zip(alist, blist):
    print(a, b)

# Output:
# a1 b1
# a2 b2
# a3 b3

alist = []
len(list(zip(alist, blist)))

# Output:
# 0
```

Pour remplir des listes de longueur inégale à la plus longue avec `None` utilisez `itertools.zip_longest` (`itertools.izip_longest` dans Python 2)

```
alist = ['a1', 'a2', 'a3']
blist = ['b1']
clist = ['c1', 'c2', 'c3', 'c4']

for a,b,c in itertools.zip_longest(alist, blist, clist):
    print(a, b, c)

# Output:
# a1 b1 c1
# a2 None c2
# a3 None c3
# None None c4
```

3. **Insérer dans un index spécifique des valeurs:**

```
 alist = [123, 'xyz', 'zara', 'abc']
 alist.insert(3, [2009])
 print("Final List :", alist)
```

Sortie:

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

Tout et tous

Vous pouvez utiliser `all()` pour déterminer si toutes les valeurs d'une itération sont évaluées sur True

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

De même, `any()` détermine si une ou plusieurs valeurs dans une itération sont évaluées sur True

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

Bien que cet exemple utilise une liste, il est important de noter que ces éléments intégrés fonctionnent avec tous les itérables, y compris les générateurs.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

Supprimer les valeurs en double dans la liste

La suppression des valeurs en double dans une liste peut être effectuée en convertissant la liste en un `set` (c'est-à-dire une collection non ordonnée d'objets distincts). Si une structure de données de `list` est nécessaire, alors l'ensemble peut être reconverti en une liste à l'aide de la `list()` fonctions `list()` :

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

Notez qu'en convertissant une liste en un ensemble, la commande d'origine est perdue.

Pour conserver l'ordre de la liste, on peut utiliser un `OrderedDict`

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

Accès aux valeurs dans la liste imbriquée

En commençant par une liste en trois dimensions:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

Accéder aux éléments de la liste:

```
print(alist[0][0][1])
#2
#Accesses second element in the first list in the first list

print(alist[1][1][2])
#10
#Accesses the third element in the second list in the second list
```

Effectuer des opérations de support:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Appends 11 to the end of the first list in the first list
```

Utilisation de boucles imbriquées pour imprimer la liste:

```
for row in alist: #One way to loop through nested lists
    for col in row:
        print(col)
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#[12, 13, 14]
```

Notez que cette opération peut être utilisée dans une compréhension de liste ou même en tant que générateur pour produire des efficacités, par exemple:

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

Tous les éléments des listes externes ne doivent pas être eux-mêmes des listes:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Une autre façon d'utiliser les boucles imbriquées. L'autre façon est meilleure mais j'ai eu besoin de l'utiliser à l'occasion:

```
for row in range(len(alist)): #A less Pythonic way to loop through lists
    for col in range(len(alist[row])):
       print(alist[row][col])
```

```
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#15
#[12, 13, 14]
```

Utilisation de tranches dans la liste imbriquée:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

La liste finale:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

Comparaison de listes

Il est possible de comparer des listes et d'autres séquences lexicographiquement en utilisant des opérateurs de comparaison. Les deux opérandes doivent être du même type.

```
[1, 10, 100] < [2, 10, 100]
# True, because 1 < 2
[1, 10, 100] < [1, 10, 100]
# False, because the lists are equal
[1, 10, 100] <= [1, 10, 100]
# True, because the lists are equal
[1, 10, 100] < [1, 10, 101]
# True, because 100 < 101
[1, 10, 100] < [0, 10, 100]
# False, because 0 < 1
```

Si l'une des listes est contenue au début de l'autre, la liste la plus courte l'emporte.

```
[1, 10] < [1, 10, 100]
# True
```

Initialisation d'une liste à un nombre fixe d'éléments

Pour les éléments **immuables** (par exemple, `None` , littéraux de chaîne, etc.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

Pour les éléments **mutables** , la même construction se traduira par tous les éléments de la liste faisant référence au même objet, par exemple, pour un ensemble:

```
>>> my_list=[{1}] * 10
```

```
>>> print(my_list)
[{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}]
>>> my_list[0].add(2)
>>> print(my_list)
[{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}]
```

Au lieu de cela, pour initialiser la liste avec un nombre fixe d'objets **différents mutables** , utilisez:

```
my_list=[{1} for _ in range(10)]
```

Lire liste en ligne: https://riptutorial.com/fr/python/topic/209/liste

Chapitre 107: Liste de coupe (sélection de parties de listes)

Syntaxe

- a [début: fin] # éléments commencent par fin-1
- a [start:] # éléments commencent dans le reste du tableau
- a [: end] # éléments du début à la fin-1
- un [début: fin: étape] # commence par pas passé, par étape
- a [:] # une copie du tableau entier
- la source

Remarques

- `lst[::-1]` vous donne une copie inversée de la liste
- `start` ou `end` peut être un nombre négatif, ce qui signifie qu'il compte à partir de la fin du tableau au lieu du début. Alors:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

(source)

Examples

Utiliser le troisième argument "step"

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

Sélection d'une sous-liste dans une liste

```
lst = ['a', 'b', 'c', 'd', 'e']

lst[2:4]
# Output: ['c', 'd']

lst[2:]
# Output: ['c', 'd', 'e']
```

```
lst[:4]
# Output: ['a', 'b', 'c', 'd']
```

Inverser une liste avec trancher

```
a = [1, 2, 3, 4, 5]

# steps through the list backwards (step=-1)
b = a[::-1]

# built-in list method to reverse 'a'
a.reverse()

if a = b:
    print(True)

print(b)

# Output:
# True
# [5, 4, 3, 2, 1]
```

Décaler une liste en utilisant le tranchage

```
def shift_list(array, s):
    """Shifts the elements of a list to the left or right.

    Args:
        array - the list to shift
        s - the amount to shift the list ('+': right-shift, '-': left-shift)

    Returns:
        shifted_array - the shifted list
    """
    # calculate actual shift amount (e.g., 11 --> 1 if length of the array is 5)
    s %= len(array)

    # reverse the shift direction to be more intuitive
    s *= -1

    # shift array with list slicing
    shifted_array = array[s:] + array[:s]

    return shifted_array

my_array = [1, 2, 3, 4, 5]

# negative numbers
shift_list(my_array, -7)
>>> [3, 4, 5, 1, 2]

# no shift on numbers equal to the size of the array
shift_list(my_array, 5)
>>> [1, 2, 3, 4, 5]

# works on positive numbers
shift_list(my_array, 3)
```

```
>>> [3, 4, 5, 1, 2]
```

Lire Liste de coupe (sélection de parties de listes) en ligne:
https://riptutorial.com/fr/python/topic/1494/liste-de-coupe--selection-de-parties-de-listes-

Chapitre 108: Liste des compréhensions

Introduction

Les compréhensions de liste en Python sont des constructions syntaxiques concises. Ils peuvent être utilisés pour générer des listes à partir d'autres listes en appliquant des fonctions à chaque élément de la liste. La section suivante explique et illustre l'utilisation de ces expressions.

Syntaxe

- [x + 1 pour x dans (1, 2, 3)] # compréhension de liste, donne [2, 3, 4]
- (x + 1 pour x dans (1, 2, 3)) # expression du générateur, donnera 2, puis 3, puis 4
- [x pour x dans (1, 2, 3) si x% 2 == 0] # liste compréhension avec filtre, donne [2]
- [x + 1 si x% 2 == 0 sinon x pour x dans (1, 2, 3)] # compréhension de liste avec ternaire
- [x + 1 si x% 2 == 0 sinon x pour x dans la plage (-3,4) si x> 0] # liste compréhension avec ternaire et filtrage
- {x pour x dans (1, 2, 2, 3)} # définir la compréhension, donne {1, 2, 3}
- {k: v pour k, v dans [('a', 1), ('b', 2)]} # dict la compréhension, donne {'a': 1, 'b': 2} (python 2.7+ et 3.0+ seulement)
- [x + y pour x dans [1, 2] pour y dans [10, 20]] # Boucles imbriquées, donne [11, 21, 12, 22]
- [x + y pour x dans [1, 2, 3] si x> 2 pour y dans [3, 4, 5]] # Condition vérifiée au 1er pour la boucle
- [x + y pour x dans [1, 2, 3] pour y dans [3, 4, 5] si x> 2] # Condition vérifiée en 2ème pour la boucle
- [x pour x dans xrange (10) si x% 2 == 0] # condition vérifiée si les nombres en boucle sont des nombres impairs

Remarques

Les compréhensions sont des constructions syntaxiques qui définissent des structures de données ou des expressions propres à un langage particulier. Une utilisation correcte des compréhensions réinterprète celles-ci en expressions faciles à comprendre. En tant qu'expressions, elles peuvent être utilisées:

- à droite des missions
- comme arguments pour les appels de fonction
- dans le corps d' une fonction lambda
- comme déclarations autonomes. (Par exemple: `[print(x) for x in range(10)]`)

Examples

Liste des compréhensions

Une compréhension de liste crée une nouvelle `list` en appliquant une expression à chaque

élément d'une itération . La forme la plus élémentaire est:

```
[ <expression> for <element> in <iterable> ]
```

Il y a aussi une condition optionnelle "if":

```
[ <expression> for <element> in <iterable> if <condition> ]
```

Chaque `<element>` dans le `<iterable>` est branché à la `<expression>` si (en option) `<condition>` évalue à true . Tous les résultats sont renvoyés immédiatement dans la nouvelle liste. Les expressions de générateur sont évaluées paresseusement, mais les compréhensions de liste évaluent immédiatement tout l'itérateur - une mémoire consommatrice proportionnelle à la longueur de l'itérateur.

Pour créer une `list` d'entiers carrés:

```
squares = [x * x for x in (1, 2, 3, 4)]
# squares: [1, 4, 9, 16]
```

L'expression `for` définit `x` sur chaque valeur à partir de `(1, 2, 3, 4)` . Le résultat de l'expression `x * x` est ajouté à une `list` interne. La `list` interne est affectée aux `squares` variables une fois remplie.

Outre une augmentation de la vitesse (comme expliqué ici), une compréhension de la liste est à peu près équivalente à la suivante pour la boucle:

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
# squares: [1, 4, 9, 16]
```

L'expression appliquée à chaque élément peut être aussi complexe que nécessaire:

```
# Get a list of uppercase characters from a string
[s.upper() for s in "Hello World"]
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']

# Strip off any commas from the end of strings in a list
[w.strip(',') for w in ['these,', 'words,,', 'mostly', 'have,commas,']]
# ['these', 'words', 'mostly', 'have,commas']

# Organize letters in words more reasonably - in an alphabetical order
sentence = "Beautiful is better than ugly"
["".join(sorted(word, key = lambda x: x.lower())) for word in sentence.split()]
# ['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

---

autre

`else` peut être utilisé dans les constructions de compréhension de liste, mais faites attention à la syntaxe. If / else clauses doivent être utilisés avant `for` la boucle, et non après:

```
# create a list of characters in apple, replacing non vowels with '*'
# Ex - 'apple' --> ['a', '*', '*', '*' ,'e']

[x for x in 'apple' if x in 'aeiou' else '*']
#SyntaxError: invalid syntax

# When using if/else together use them before the loop
[x if x in 'aeiou' else '*' for x in 'apple']
#['a', '*', '*', '*', 'e']
```

Notez que cela utilise une construction de langage différente, une expression conditionnelle , qui ne fait pas partie de la syntaxe de compréhension . Alors que `if` après le `for…in` *fait* partie des compréhensions de listes et permet de *filtrer des* éléments de la source itérable.

Double itération

L'ordre de la double itération `[... for x in ... for y in ...]` est soit naturel, soit contre-intuitif. La règle de base est de suivre un équivalent `for` boucle:

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```

Cela devient:

```
[str(x)
    for i in range(3)
        for x in foo(i)
]
```

Cela peut être compressé en une seule ligne comme `[str(x) for i in range(3) for x in foo(i)]`

Mutation en place et autres effets secondaires

Avant d'utiliser la compréhension de liste, comprenez la différence entre les fonctions appelées pour leurs effets secondaires (fonctions *mutantes* ou *in-* situ) qui renvoient généralement `None` et les fonctions qui renvoient une valeur intéressante.

De nombreuses fonctions (en particulier *les* fonctions *pures*) prennent simplement un objet et

renvoient un objet. Une fonction sur *place* modifie l'objet existant, appelé *effet secondaire* . D'autres exemples incluent des opérations d'entrée et de sortie telles que l'impression.

`list.sort()` trie une liste *sur place* (ce qui signifie qu'elle modifie la liste d'origine) et renvoie la valeur `None` . Par conséquent, cela ne fonctionnera pas comme prévu dans une liste de compréhension:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

Au lieu de cela, `sorted()` renvoie une `list` triée plutôt que de trier sur place:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

L'utilisation de la compréhension des effets secondaires est possible, comme les fonctions d'E / S ou sur place. Pourtant, une boucle for est généralement plus lisible. Alors que cela fonctionne dans Python 3:

```
[print(x) for x in (1, 2, 3)]
```

Au lieu de cela, utilisez:

```
for x in (1, 2, 3):
    print(x)
```

Dans certaines situations, les fonctions d'effets secondaires *sont* adaptés à la compréhension de la liste. `random.randrange()` a pour effet secondaire de modifier l'état du générateur de nombres aléatoires, mais il renvoie également une valeur intéressante. En outre, `next()` peut être appelé sur un itérateur.

Le générateur de valeur aléatoire suivant n'est pas pur, mais il a du sens car le générateur aléatoire est réinitialisé à chaque fois que l'expression est évaluée:

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

Les espaces dans les listes compréhensibles

Des compréhensions de listes plus compliquées peuvent atteindre une durée indésirable ou devenir moins lisibles. Bien que moins courant dans les exemples, il est possible de diviser une compréhension de liste en plusieurs lignes comme suit:

```
[
    x for x
```

```
    in 'foo'
    if x not in 'bar'
]
```

Compréhensions du dictionnaire

Une compréhension de dictionnaire est similaire à une compréhension de liste sauf qu'elle produit un objet dictionnaire au lieu d'une liste.

Un exemple de base:

Python 2.x 2.7

```
{x: x * x for x in (1, 2, 3, 4)}
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

ce qui est juste une autre façon d'écrire:

```
dict((x, x * x) for x in (1, 2, 3, 4))
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

Comme avec une compréhension de liste, nous pouvons utiliser une instruction conditionnelle dans la compréhension du dict pour produire uniquement les éléments dict répondant à certains critères.

Python 2.x 2.7

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# Out: {'Exchange': 8, 'Overflow': 8}
```

Ou, réécrit en utilisant une expression de générateur.

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# Out: {'Exchange': 8, 'Overflow': 8}
```

Commencer par un dictionnaire et utiliser la compréhension du dictionnaire comme filtre de paire clé-valeur

Python 2.x 2.7

```
initial_dict = {'x': 1, 'y': 2}
{key: value for key, value in initial_dict.items() if key == 'x'}
# Out: {'x': 1}
```

Changement de clé et valeur du dictionnaire (dictionnaire inversé)

Si vous avez un dict contenant des valeurs *hashable* simples (les valeurs en double peuvent avoir des résultats inattendus):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

et vous vouliez échanger les clés et les valeurs, vous pouvez prendre plusieurs approches en fonction de votre style de codage:

- `swapped = {v: k for k, v in my_dict.items()}`
- `swapped = dict((v, k) for k, v in my_dict.iteritems())`
- `swapped = dict(zip(my_dict.values(), my_dict))`
- `swapped = dict(zip(my_dict.values(), my_dict.keys()))`
- `swapped = dict(map(reversed, my_dict.items()))`

```
print(swapped)
# Out: {a: 1, b: 2, c: 3}
```

Python 2.x 2.3

Si votre dictionnaire est volumineux, envisagez d' *importer des outils itert* et d'utiliser `izip` ou `imap` .

Fusion de dictionnaires

Combinez les dictionnaires et remplacez éventuellement les anciennes valeurs par une compréhension imbriquée du dictionnaire.

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Cependant, le déballage du dictionnaire (PEP 448) peut être préféré.

Python 3.x 3.5

```
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Remarque : les définitions de dictionnaires ont été ajoutées à Python 3.0 et transférées vers les versions 2.7+, contrairement aux interprétations de listes ajoutées en 2.0. Les versions <2.7 peuvent utiliser des expressions de générateur et le `dict()` pour simuler le comportement des définitions de dictionnaires.

Expressions de générateur

Les expressions de générateur sont très similaires aux compréhensions de liste. La principale différence est que cela ne crée pas un ensemble complet de résultats à la fois; il crée un objet générateur qui peut ensuite être itéré.

Par exemple, voyez la différence dans le code suivant:

```
# list comprehension
```

```
[x**2 for x in range(10)]
# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python 2.x 2.4

```
# generator comprehension
(x**2 for x in xrange(10))
# Output: <generator object <genexpr> at 0x11b4b7c80>
```

Ce sont deux objets très différents:

- la compréhension de la liste renvoie un objet `list` alors que la compréhension du générateur renvoie un `generator` .
- `generator` objets `generator` ne peuvent pas être indexés et utilisent la fonction `next` pour récupérer les éléments.

Note : Nous utilisons `xrange` car il crée aussi un objet générateur. Si nous utilisions la plage, une liste serait créée. De plus, `xrange` n'existe que dans la version ultérieure de python 2. Dans python 3, `range` ne fait que renvoyer un générateur. Pour plus d'informations, reportez-vous à l' exemple des *fonctions Différences entre plage et xrange* .

Python 2.x 2.4

```
g = (x**2 for x in xrange(10))
print(g[0])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object has no attribute '__getitem__'
```

```
g.next()  # 0
g.next()  # 1
g.next()  # 4
...
g.next()  # 81

g.next()  # Throws StopIteration Exception
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Python 3.x 3.0

> REMARQUE: La fonction `g.next()` doit être remplacée par `next(g)` et `xrange` avec `range` car `Iterator.next()` et `xrange()` n'existent pas dans Python 3.

Bien que les deux puissent être répétés de la même manière:

```
for i in [x**2 for x in range(10)]:
    print(i)

"""
Out:
0
1
4
...
81
"""
```

Python 2.x 2.4

```
for i in (x**2 for x in xrange(10)):
    print(i)

"""
Out:
0
1
4
.
.
.
81
"""
```

Cas d'utilisation

Les expressions de générateur sont évaluées paresseusement, ce qui signifie qu'elles génèrent et renvoient chaque valeur uniquement lorsque le générateur est itéré. Ceci est souvent utile lorsque vous parcourez des jeux de données volumineux, évitant de créer un doublon du jeu de données en mémoire:

```
for square in (x**2 for x in range(1000000)):
    #do something
```

Un autre cas d'utilisation courant consiste à éviter de procéder à une itération complète si cela n'est pas nécessaire. Dans cet exemple, un élément est extrait d'une API distante à chaque itération de `get_objects()` . Des milliers d'objets peuvent exister, doivent être récupérés un par un et il suffit de savoir si un objet correspondant à un modèle existe. En utilisant une expression de générateur, lorsque nous rencontrons un objet correspondant au motif.

```
def get_objects():
    """Gets objects from an API one by one"""
    while True:
        yield get_next_item()

def object_matches_pattern(obj):
    # perform potentially complex calculation
    return matches_pattern
```

```
def right_item_exists():
    items = (object_matched_pattern(each) for each in get_objects())
    for item in items:
        if item.is_the_right_one:


            return True
    return False
```

Définir les compréhensions

La compréhension d'ensemble est similaire à la compréhension de liste et du dictionnaire , mais elle produit un ensemble , qui est une collection non ordonnée d'éléments uniques.

Python 2.x 2.7

```
# A set containing every value in range(5):
{x for x in range(5)}
# Out: {0, 1, 2, 3, 4}

# A set of even numbers between 1 and 10:
{x for x in range(1, 11) if x % 2 == 0}
# Out: {2, 4, 6, 8, 10}

# Unique alphabetic characters in a string of text:
text = "When in the Course of human events it becomes necessary for one people..."
{ch.lower() for ch in text if ch.isalpha()}
# Out: set(['a', 'c', 'b', 'e', 'f', 'i', 'h', 'm', 'l', 'o',
#           'n', 'p', 's', 'r', 'u', 't', 'w', 'v', 'y'])
```

Démo en direct

Gardez à l'esprit que les décors ne sont pas ordonnés. Cela signifie que l'ordre des résultats dans l'ensemble peut différer de celui présenté dans les exemples ci-dessus.

Remarque : La compréhension d'ensemble est disponible depuis python 2.7+, contrairement aux compréhensions de liste, qui ont été ajoutées en 2.0. Dans Python 2.2 à Python 2.6, la fonction `set()` peut être utilisée avec une expression de générateur pour produire le même résultat:

Python 2.x 2.2

```
set(x for x in range(5))
# Out: {0, 1, 2, 3, 4}
```

Eviter les opérations répétitives et coûteuses en utilisant une clause conditionnelle

Considérez la compréhension de la liste ci-dessous:

```
>>> def f(x):
...     import time
...     time.sleep(.1)       # Simulate expensive function
...     return x**2
```

```
>>> [f(x) for x in range(1000) if f(x) > 10]
[16, 25, 36, ...]
```

Cela se traduit par deux appels à `f(x)` pour 1 000 valeurs de `x` : un appel pour générer la valeur et l'autre pour vérifier la condition `if` . Si `f(x)` est une opération particulièrement coûteuse, cela peut avoir des conséquences significatives sur les performances. Pire encore, si appeler `f()` a des effets secondaires, cela peut avoir des résultats surprenants.

Au lieu de cela, vous devez évaluer l'opération coûteuse une seule fois pour chaque valeur de `x` en générant une itération intermédiaire (expression de générateur) comme suit:

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

Ou, en utilisant l'équivalent de la carte intégrée:

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

Une autre façon d'obtenir un code plus lisible consiste à placer le résultat partiel (`v` dans l'exemple précédent) dans une itération (telle qu'une liste ou un tuple), puis à l'itérer. Puisque `v` sera le seul élément de l'itérable, le résultat est que nous avons maintenant une référence à la sortie de notre fonction lente calculée une seule fois:

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
[16, 25, 36, ...]
```

Cependant, dans la pratique, la logique du code peut être plus compliquée et il est important de la garder lisible. En général, une fonction de générateur séparée est recommandée par rapport à une ligne simple complexe:

```
>>> def process_prime_numbers(iterable):
...     for x in iterable:
...         if is_prime(x):
...             yield f(x)
...
>>> [x for x in process_prime_numbers(range(1000)) if x > 10]
[11, 13, 17, 19, ...]
```

Une autre façon d'éviter le calcul de `f(x)` plusieurs fois est d'utiliser le `@functools.lru_cache()` (Python 3.2+) décorateur de `f(x)` . De cette façon, puisque la sortie de `f` pour l'entrée `x` a déjà été calculée une fois, l'invocation de la seconde fonction de la compréhension de la liste d'origine sera aussi rapide qu'une recherche dans un dictionnaire. Cette approche utilise la mémorisation pour améliorer l'efficacité, ce qui est comparable à l'utilisation des expressions génératrices.

Dites que vous devez aplatir une liste

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Certaines des méthodes pourraient être:

```
reduce(lambda x, y: x+y, l)

sum(l, [])

list(itertools.chain(*l))
```

Cependant, la compréhension de la liste fournirait la meilleure complexité temporelle.

```
[item for sublist in l for item in sublist]
```

Les raccourcis basés sur + (y compris l'utilisation implicite en somme) sont, par nécessité, O (L ^ 2) quand il ya L sous-listes - à mesure que la liste des résultats intermédiaires s'allonge, à chaque étape allouée, et tous les éléments du résultat intermédiaire précédent doivent être copiés (ainsi que quelques nouveaux ajoutés à la fin). Donc, pour des raisons de simplicité et sans perte de généralité, disons que vous avez L sous-listes de I articles chacun: les I premiers objets sont copiés en arrière L-1 fois, le second I éléments L-2 fois, et ainsi de suite; le nombre total de copies est I fois la somme de x pour x de 1 à L exclu, c'est-à-dire I * (L ** 2) / 2.

La compréhension de la liste génère une seule liste, une fois, et copie chaque élément (de son lieu de résidence d'origine à la liste de résultats) également une seule fois.

Compréhensions impliquant des tuples

La clause `for` d'une compréhension de liste peut spécifier plus d'une variable:

```
[x + y for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [3, 7, 11]

[x + y for x, y in zip([1, 3, 5], [2, 4, 6])]
# Out: [3, 7, 11]
```

C'est comme `for` boucles régulières:

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

Notez cependant que si l'expression qui commence la compréhension est un tuple, alors il faut la parenthèses:

```
[x, y for x, y in [(1, 2), (3, 4), (5, 6)]]
# SyntaxError: invalid syntax

[(x, y) for x, y in [(1, 2), (3, 4), (5, 6)]]
```

```
# Out: [(1, 2), (3, 4), (5, 6)]
```

Compter les occurrences en utilisant la compréhension

Lorsque nous voulons compter le nombre d'éléments d'une itération, qui remplissent certaines conditions, nous pouvons utiliser la compréhension pour produire une syntaxe idiomatique:

```
# Count the numbers in `range(1000)` that are even and contain the digit `9`:
print (sum(
    1 for x in range(1000)
    if x % 2 == 0 and
    '9' in str(x)
))
# Out: 95
```

Le concept de base peut être résumé comme suit:

1. Itérer sur les éléments de la `range(1000)` .
2. Concaténer tous les nécessaires `if` les conditions.
3. Utilisez 1 comme *expression* pour renvoyer un 1 pour chaque élément répondant aux conditions.
4. Résumez tous les `1` s pour déterminer le nombre d'éléments répondant aux conditions.

Note : Ici, nous ne collectons pas les `1` dans une liste (notez l'absence de crochets), mais nous les transmettons directement à la fonction `sum` qui les additionne. Ceci est appelé une *expression de générateur* , qui est similaire à une compréhension.

Modification de types dans une liste

Les données quantitatives sont souvent lues comme des chaînes devant être converties en types numériques avant le traitement. Les types de tous les éléments de la liste peuvent être convertis avec une fonction List Comprehension ou la fonction `map()` .

```
# Convert a list of strings to integers.
items = ["1","2","3","4"]
[int(item) for item in items]
# Out: [1, 2, 3, 4]

# Convert a list of strings to float.
items = ["1","2","3","4"]
map(float, items)
# Out:[1.0, 2.0, 3.0, 4.0]
```

Lire Liste des compréhensions en ligne: https://riptutorial.com/fr/python/topic/196/liste-des-comprehensions

Chapitre 109: Liste des compréhensions

Introduction

Une compréhension de liste est un outil syntaxique pour créer des listes de manière naturelle et concise, comme illustré dans le code suivant pour faire une liste de carrés des nombres 1 à 10: `[i ** 2 for i in range(1,11)]` Le dummy `i` d'une `range` liste existante est utilisé pour créer un nouveau motif d'élément. Il est utilisé là où une boucle for serait nécessaire dans des langages moins expressifs.

Syntaxe

- [i pour i in range (10)] # compréhension de base de la liste
- [i for i in xrange (10)] # compréhension de base de la liste avec objet générateur en python 2.x
- [i pour i dans la plage (20) si i% 2 == 0] # avec le filtre
- [x + y pour x dans [1, 2, 3] pour y dans [3, 4, 5]] # boucles imbriquées
- [i si i> 6 sinon 0 pour i dans la plage (10)] # expression ternaire
- [i si i> 4 sinon 0 pour i dans la plage (20) si i% 2 == 0] # avec filtre et expression ternaire
- [[x + y pour x dans [1, 2, 3]] pour y dans [3, 4, 5]] # compréhension de liste imbriquée

Remarques

Les compréhensions de liste ont été décrites dans PEP 202 et introduites dans Python 2.0.

Examples

Liste conditionnelle

Étant donné une compréhension de la liste, vous pouvez ajouter une ou plusieurs conditions `if` pour filtrer les valeurs.

```
[<expression> for <element> in <iterable> if <condition>]
```

Pour chaque `<element>` dans `<iterable>` ; Si `<condition>` `True` , ajoutez `<expression>` (généralement une fonction de `<element>`) à la liste renvoyée.

Par exemple, cela peut être utilisé pour extraire uniquement des nombres pairs d'une suite d'entiers:

```
[x for x in range(10) if x % 2 == 0]
# Out: [0, 2, 4, 6, 8]
```

Démo en direct

Le code ci-dessus est équivalent à:

```
even_numbers = []
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# Out: [0, 2, 4, 6, 8]
```

En outre, une compréhension conditionnelle de la forme `[e for x in y if c]` (où `e` et `c` sont des expressions en termes de `x`) est équivalente à `list(filter(lambda x: c, map(lambda x: e, y)))` .

Malgré le même résultat, faites attention au fait que le premier exemple est presque deux fois plus rapide que le second. Pour ceux qui sont curieux, c'est une bonne explication de la raison pour laquelle.

Notez que ceci est très différent de l'expression conditionnelle `... if ... else ...` (parfois appelée expression ternaire) que vous pouvez utiliser pour la partie `<expression>` de la compréhension de liste. Prenons l'exemple suivant:

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

Démo en direct

Ici, l'expression conditionnelle n'est pas un filtre, mais plutôt un opérateur déterminant la valeur à utiliser pour les éléments de la liste:

```
<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>
```

Cela devient plus évident si vous le combinez avec d’autres opérateurs:

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in range(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Démo en direct

Si vous utilisez Python 2.7, `xrange` peut être préférable à `range` pour plusieurs raisons, comme décrit dans la documentation de `xrange` .

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in xrange(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Le code ci-dessus est équivalent à:

```
numbers = []
```

```
for x in range(10):
    if x % 2 == 0:
        temp = x
    else:
        temp = -1
    numbers.append(2 * temp + 1)
print(numbers)
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

On peut combiner des expressions ternaires et `if` conditions. L'opérateur ternaire travaille sur le résultat filtré:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

La même chose n'aurait pas pu être réalisée uniquement par l'opérateur ternaire:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out:['*', '*', '*', '*', 4, '*', 6, '*', 8, '*']
```

Voir aussi: Filtres , qui offrent souvent une alternative suffisante aux compréhensions de listes conditionnelles.

Liste des compréhensions avec des boucles imbriquées

Liste compréhensions peuvent utiliser imbriqués `for` les boucles. Vous pouvez coder un nombre quelconque de boucles imbriquées pour l' intérieur d' une compréhension de la liste, et chaque `for` la boucle peut avoir une option associée `if` test. Ce faisant, l'ordre du `for` des constructions est le même ordre que lors de l' écriture d' une série d'imbriquée `for` les déclarations. La structure générale des compréhensions de listes ressemble à ceci:

```
[ expression for target1 in iterable1 [if condition1]
              for target2 in iterable2 [if condition2]...
              for targetN in iterableN [if conditionN] ]
```

Par exemple, le code suivant aplatit une liste de listes utilisant plusieurs `for` instructions:

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

peut être écrit comme une compréhension équivalente de liste avec de multiples `for` les constructions:

```
data = [[1, 2], [3, 4], [5, 6]]
```

```
output = [element for each_list in data for element in each_list]
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

Démo en direct

À la fois dans la forme développée et dans la compréhension de la liste, la boucle externe (la première pour l'instruction) vient en premier.

En plus d'être plus compact, la compréhension imbriquée est également beaucoup plus rapide.

```
In [1]: data = [[1,2],[3,4],[5,6]]
In [2]: def f():
   ...:     output=[]
   ...:     for each_list in data:
   ...:         for element in each_list:
   ...:             output.append(element)
   ...:     return output
In [3]: timeit f()
1000000 loops, best of 3: 1.37 µs per loop
In [4]: timeit [inner for outer in data for inner in outer]
1000000 loops, best of 3: 632 ns per loop
```

La surcharge pour l'appel de fonction ci-dessus est d'environ *140ns* .

En ligne `if` s sont imbriquées de la même manière, et peuvent se produire dans n'importe quelle position après le premier `for` :

```
data = [[1], [2, 3], [4, 5]]
output = [element for each_list in data
                if len(each_list) == 2
                for element in each_list
                if element != 5]
print(output)
# Out: [2, 3, 4]
```

Démo en direct

Par souci de lisibilité, cependant, vous devriez envisager d'utiliser des *boucles* traditionnelles. Cela est particulièrement vrai lorsque l'imbrication a plus de 2 niveaux de profondeur et / ou que la logique de la compréhension est trop complexe. la compréhension multiple de la liste de boucles imbriquées pourrait être source d'erreurs ou donner des résultats inattendus.

Filtre de refactoring et carte pour lister les compréhensions

Les fonctions de `filter` ou de `map` doivent souvent être remplacées par des listes compréhensibles . Guido Van Rossum le décrit bien dans une lettre ouverte en 2005 :

> `filter(P, S)` est presque toujours écrit plus clair que `[x for x in S if P(x)]` , ce qui a l'énorme avantage que les usages les plus courants impliquent prédicats des

comparaisons, par exemple `x==42` , et définissant un lambda pour cela nécessite juste beaucoup plus d'effort pour le lecteur (plus le lambda est plus lent que la compréhension de la liste). Encore plus pour la `map(F, S)` qui devient `[F(x) for x in S]` . Bien sûr, dans de nombreux cas, vous pourriez utiliser des expressions de générateur à la place.

Les lignes de code suivantes sont considérées comme " *non pythoniques* " et provoquent des erreurs dans de nombreux *linters* en python.

```
filter(lambda x: x % 2 == 0, range(10)) # even numbers < 10
map(lambda x: 2*x, range(10)) # multiply each number by two
reduce(lambda x,y: x+y, range(10)) # sum of all elements in list
```

En prenant ce que nous avons appris de la citation précédente, nous pouvons décomposer ces expressions de `filter` et de `map` dans leurs *compréhensions de liste* équivalentes; aussi supprimer les fonctions *lambda* de chaque - rendant le code plus lisible dans le processus.

```
# Filter:
# P(x) = x % 2 == 0
# S = range(10)
[x for x in range(10) if x % 2 == 0]

# Map
# F(x) = 2*x
# S = range(10)
[2*x for x in range(10)]
```

La lisibilité devient encore plus évidente dans le cas des fonctions de chaînage. En raison de la lisibilité, les résultats d'une carte ou d'une fonction de filtre doivent être transmis à la suite; avec des cas simples, ceux-ci peuvent être remplacés par une seule compréhension de liste. De plus, nous pouvons facilement comprendre, à partir de la compréhension de la liste, le résultat de notre processus, où il y a plus de charge cognitive lors du raisonnement sur le processus Map & Filter enchaîné.

```
# Map & Filter
filtered = filter(lambda x: x % 2 == 0, range(10))
results = map(lambda x: 2*x, filtered)

# List comprehension
results = [2*x for x in range(10) if x % 2 == 0]
```

Refactoring - Référence rapide

- **Carte**

  ```
  map(F, S) == [F(x) for x in S]
  ```

- **Filtre**

```
filter(P, S) == [x for x in S if P(x)]
```

où `F` et `P` sont des fonctions qui transforment respectivement les valeurs d'entrée et renvoient un `bool`

Compréhension de liste imbriquée

Les compréhensions de liste imbriquées, contrairement aux compréhensions de liste avec des boucles imbriquées, sont des compréhensions de liste dans une compréhension de liste. L'expression initiale peut être n'importe quelle expression arbitraire, y compris une autre compréhension de la liste.

```
#List Comprehension with nested loop
[x + y for x in [1, 2, 3] for y in [3, 4, 5]]
#Out: [4, 5, 6, 5, 6, 7, 6, 7, 8]

#Nested List Comprehension
[[x + y for x in [1, 2, 3]] for y in [3, 4, 5]]
#Out: [[4, 5, 6], [5, 6, 7], [6, 7, 8]]
```

L'exemple imbriqué est équivalent à

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

Un exemple où une compréhension imbriquée peut être utilisée pour transposer une matrice.

```
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]

[[row[i] for row in matrix] for i in range(len(matrix))]
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Comme `for` boucles imbriquées, il n'y a pas de limite à la façon dont les compréhensions profondes peuvent être imbriquées.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [[['1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

Itérer deux ou plusieurs listes simultanément dans la compréhension de liste

Pour itérer plus de deux listes simultanément dans la *compréhension de la liste* , on peut utiliser `zip()` comme:

```
>>> list_1 = [1, 2, 3 , 4]
>>> list_2 = ['a', 'b', 'c', 'd']
```

```
>>> list_3 = ['6', '7', '8', '9']

# Two lists
>>> [(i, j) for i, j in zip(list_1, list_2)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# Three lists
>>> [(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]

# so on ...
```

Lire Liste des compréhensions en ligne: https://riptutorial.com/fr/python/topic/5265/liste-des-comprehensions

Chapitre 110: Listes liées

Introduction

Une liste chaînée est une collection de nœuds, chacun composé d'une référence et d'une valeur. Les nœuds sont assemblés en une séquence en utilisant leurs références. Les listes liées peuvent être utilisées pour implémenter des structures de données plus complexes telles que les listes, les piles, les files d'attente et les tableaux associatifs.

Examples

Exemple de liste liée unique

Cet exemple implémente une liste liée avec plusieurs des mêmes méthodes que celle de l'objet de liste intégré.

```
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, val):
        self.data = val

    def setNext(self, val):
        self.next = val

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        """Check if the list is empty"""
        return self.head is None

    def add(self, item):
        """Add the item to the list"""
        new_node = Node(item)
        new_node.setNext(self.head)
        self.head = new_node

    def size(self):
        """Return the length/size of the list"""
        count = 0
        current = self.head
        while current is not None:
            count += 1
```

```
            current = current.getNext()
        return count

    def search(self, item):
        """Search for item in list. If found, return True. If not found, return False"""
        current = self.head
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
        return found

    def remove(self, item):
        """Remove item from list. If item is not found in list, raise ValueError"""
        current = self.head
        previous = None
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if found:
            if previous is None:
                self.head = current.getNext()
            else:
                previous.setNext(current.getNext())
        else:
            raise ValueError
            print 'Value not found.'

    def insert(self, position, item):
        """
        Insert item at position specified. If position specified is
        out of bounds, raise IndexError
        """
        if position > self.size() - 1:
            raise IndexError
            print "Index out of bounds."
        current = self.head
        previous = None
        pos = 0
        if position is 0:
            self.add(item)
        else:
            new_node = Node(item)
            while pos < position:
                pos += 1
                previous = current
                current = current.getNext()
            previous.setNext(new_node)
            new_node.setNext(current)

    def index(self, item):
        """
        Return the index where item is found.
        If item is not found, return None.
        """
```

```
        current = self.head
        pos = 0
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
                pos += 1
        if found:
            pass
        else:
            pos = None
        return pos

    def pop(self, position = None):
        """
        If no argument is provided, return and remove the item at the head.
        If position is provided, return and remove the item at that position.
        If index is out of bounds, raise IndexError
        """
        if position > self.size():
            print 'Index out of bounds'
            raise IndexError

        current = self.head
        if position is None:
            ret = current.getData()
            self.head = current.getNext()
        else:
            pos = 0
            previous = None
            while pos < position:
                previous = current
                current = current.getNext()
                pos += 1
                ret = current.getData()
            previous.setNext(current.getNext())
        print ret
        return ret

    def append(self, item):
        """Append item to the end of the list"""
        current = self.head
        previous = None
        pos = 0
        length = self.size()
        while pos < length:
            previous = current
            current = current.getNext()
            pos += 1
        new_node = Node(item)
        if previous is None:
            new_node.setNext(current)
            self.head = new_node
        else:
            previous.setNext(new_node)

    def printList(self):
        """Print the list"""
        current = self.head
```

```
        while current is not None:
            print current.getData()
            current = current.getNext()
```

L'utilisation fonctionne comme celle de la liste intégrée.

```
ll = LinkedList()
ll.add('l')
ll.add('H')
ll.insert(1,'e')
ll.append('l')
ll.append('o')
ll.printList()

H
e
l
l
o
```

Lire Listes liées en ligne: https://riptutorial.com/fr/python/topic/9299/listes-liees

Chapitre 111: Manipulation de XML

Remarques

Tous les éléments de l'entrée XML ne seront pas considérés comme des éléments de l'arborescence analysée. Actuellement, ce module ignore les commentaires XML, les instructions de traitement et les déclarations de type de document dans l'entrée. Néanmoins, les arborescences créées à l'aide de l'API de ce module plutôt que l'analyse à partir de texte XML peuvent contenir des commentaires et des instructions de traitement. ils seront inclus lors de la génération de sortie XML.

Examples

Ouvrir et lire en utilisant un ElementTree

Importez l'objet ElementTree, ouvrez le fichier .xml approprié et obtenez la balise racine:

```
import xml.etree.ElementTree as ET
tree = ET.parse("yourXMLfile.xml")
root = tree.getroot()
```

Il y a plusieurs façons de chercher dans l'arborescence. Le premier est par itération:

```
for child in root:
    print(child.tag, child.attrib)
```

Sinon, vous pouvez référencer des emplacements spécifiques comme une liste:

```
print(root[0][1].text)
```

Pour rechercher des balises spécifiques par nom, utilisez le `.find` ou `.findall` :

```
print(root.findall("myTag"))
print(root[0].find("myOtherTag"))
```

Modification d'un fichier XML

Importer le module Element Tree et ouvrir le fichier xml, obtenir un élément xml

```
import xml.etree.ElementTree as ET
tree = ET.parse('sample.xml')
root=tree.getroot()
element = root[0] #get first child of root element
```

L'objet Element peut être manipulé en modifiant ses champs, en ajoutant et en modifiant des attributs, en ajoutant et en supprimant des enfants

```
element.set('attribute_name', 'attribute_value') #set the attribute to xml element
element.text="string_text"
```

Si vous voulez supprimer un élément, utilisez la méthode Element.remove ()

```
root.remove(element)
```

ElementTree.write () méthode utilisée pour générer un objet xml dans des fichiers xml.

```
tree.write('output.xml')
```

Créer et créer des documents XML

Module d'importation d'éléments

```
import xml.etree.ElementTree as ET
```

La fonction Element () permet de créer des éléments XML

```
p=ET.Element('parent')
```

Fonction SubElement () utilisée pour créer des sous-éléments à un élément donner

```
c = ET.SubElement(p, 'child1')
```

La fonction dump () permet de vider des éléments XML.

```
ET.dump(p)
# Output will be like this
#<parent><child1 /></parent>
```

Si vous souhaitez enregistrer dans un fichier, créez un arbre xml avec la fonction ElementTree () et enregistrez-le dans un fichier en utilisant la méthode write ()

```
tree = ET.ElementTree(p)
tree.write("output.xml")
```

La fonction Comment () est utilisée pour insérer des commentaires dans un fichier xml.

```
comment = ET.Comment('user comment')
p.append(comment) #this comment will be appended to parent element
```

Ouverture et lecture de fichiers XML volumineux à l'aide d'iterparse (analyse incrémentielle)

Parfois, nous ne voulons pas charger l'intégralité du fichier XML pour obtenir les informations dont nous avons besoin. Dans ces cas, il est utile de pouvoir charger progressivement les sections

pertinentes et de les supprimer lorsque nous avons terminé. Avec la fonction iterparse, vous pouvez éditer l'arborescence d'éléments stockée lors de l'analyse du XML.

Importez l'objet ElementTree:

```
import xml.etree.ElementTree as ET
```

Ouvrez le fichier .xml et parcourez tous les éléments:

```
for event, elem in ET.iterparse("yourXMLfile.xml"):
    ... do something ...
```

Alternativement, nous ne pouvons rechercher que des événements spécifiques, tels que les balises de début / fin ou les espaces de noms. Si cette option est omise (comme ci-dessus), seuls les événements "fin" sont renvoyés:

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
    ... do something ...
```

Voici l'exemple complet montrant comment effacer des éléments de l'arborescence en mémoire lorsque nous en avons fini avec:

```
for event, elem in ET.iterparse("yourXMLfile.xml", events=("start","end")):
    if elem.tag == "record_tag" and event == "end":
        print elem.text
        elem.clear()
    ... do something else ...
```

Recherche du XML avec XPath

À partir de la version 2.7, `ElementTree` prend mieux en charge les requêtes XPath. XPath est une syntaxe permettant de naviguer dans un fichier XML, car SQL est utilisé pour rechercher dans une base de données. Les deux fonctions `find` et `findall` prennent en charge XPath. Le xml ci-dessous sera utilisé pour cet exemple

```
 <Catalog>
    <Books>
        <Book id="1" price="7.95">
            <Title>Do Androids Dream of Electric Sheep?</Title>
            <Author>Philip K. Dick</Author>
        </Book>
        <Book id="5" price="5.95">
            <Title>The Colour of Magic</Title>
            <Author>Terry Pratchett</Author>
        </Book>
        <Book id="7" price="6.95">
            <Title>The Eye of The World</Title>
            <Author>Robert Jordan</Author>
        </Book>
    </Books>
</Catalog>
```

Recherche de tous les livres:

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

Recherche du livre intitulé "The Color of Magic":

```
tree.find("Books/Book[Title='The Colour of Magic']")
# always use '' in the right side of the comparison
```

Recherche du livre avec id = 5:

```
tree.find("Books/Book[@id='5']")
# searches with xml attributes must have '@' before the name
```

Rechercher le deuxième livre:

```
tree.find("Books/Book[2]")
# indexes starts at 1, not 0
```

Recherchez le dernier livre:

```
tree.find("Books/Book[last()]")
# 'last' is the only xpath function allowed in ElementTree
```

Rechercher tous les auteurs:

```
tree.findall(".//Author")
#searches with // must use a relative path
```

Lire Manipulation de XML en ligne: https://riptutorial.com/fr/python/topic/479/manipulation-de-xml

Chapitre 112: Mathématiques complexes

Syntaxe

- cmath.rect (AbsoluteValue, Phase)

Examples

Arithmétique complexe avancée

Le module `cmath` inclut des fonctions supplémentaires pour utiliser des nombres complexes.

```
import cmath
```

Ce module peut calculer la phase d'un nombre complexe, en radians:

```
z = 2+3j # A complex number
cmath.phase(z) # 0.982793723247329
```

Il permet la conversion entre les représentations cartésiennes (rectangulaires) et polaires des nombres complexes:

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)
cmath.rect(2, cmath.pi/2) # (0+2j)
```

Le module contient la version complexe de

- Fonctions exponentielles et logarithmiques (comme d'habitude `log` est le logarithme naturel et `log10` le logarithme décimal):

  ```
  cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)
  cmath.log(z) # (1.2824746787307684+0.9827937232473291j)
  cmath.log10(-100) # (2+1.3643763538418412j)
  ```

- Racines carrées:

  ```
  cmath.sqrt(z) # (1.6741492280355401+0.8959774761298381j)
  ```

- Fonctions trigonométriques et leurs inverses:

  ```
  cmath.sin(z)  # (9.15449914691143-4.168906959966565j)
  cmath.cos(z)  # (-4.189625690968807-9.109227893755337j)
  cmath.tan(z)  # (-0.003764025641504249+1.003238627353611j)
  cmath.asin(z) # (0.5706527843210994+1.9833870299165355j)
  cmath.acos(z) # (1.0001435424737972-1.9833870299165355j)
  cmath.atan(z) # (1.4099210495965755+0.22907268296853878j)
  cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
  ```

- Fonctions hyperboliques et leurs inverses:

```
cmath.sinh(z)  # (-3.59056458998578+0.5309210862485197j)
cmath.cosh(z)  # (-3.7245455049153224+0.5118225699873846j)
cmath.tanh(z)  # (0.965385879022133-0.009884375038322495j)
cmath.asinh(z) # (0.5706527843210994+1.9833870299165355j)
cmath.acosh(z) # (1.9833870299165355+1.0001435424737972j)
cmath.atanh(z) # (0.14694666622552977+1.3389725222944935j)
cmath.cosh(z)**2 - cmath.sin(z)**2  # (1+0j)
cmath.cosh((0+1j)*z) - cmath.cos(z) # 0j
```

Arithmétique complexe de base

Python prend en charge l'arithmétique complexe. L'unité imaginaire est notée `j` :

```
z = 2+3j # A complex number
w = 1-7j # Another complex number
```

Les nombres complexes peuvent être additionnés, soustraits, multipliés, divisés et exponentiés:

```
z + w # (3-4j)
z - w # (1+10j)
z * w # (23-11j)
z / w # (-0.38+0.34j)
z**3  # (-46+9j)
```

Python peut également extraire les parties réelles et imaginaires des nombres complexes et calculer leur valeur absolue et conjuguer:

```
z.real # 2.0
z.imag # 3.0
abs(z) # 3.605551275463989
z.conjugate() # (2-3j)
```

Lire Mathématiques complexes en ligne:
https://riptutorial.com/fr/python/topic/1142/mathematiques-complexes

Chapitre 113: Métaclasses

Introduction

Les métaclasses vous permettent de modifier en profondeur le comportement des classes Python (en termes de définition, d'instanciation, d'accès, etc.) en remplaçant la métaclasse de `type` utilisée par défaut par les nouvelles classes.

Remarques

Lors de la conception de votre architecture, considérez que de nombreuses choses peuvent être accomplies avec des métaclasses en utilisant une sémantique plus simple:

- L'héritage traditionnel est souvent plus que suffisant.
- Les décorateurs de classes peuvent intégrer des fonctionnalités dans une classe selon une approche ad hoc.
- Python 3.6 introduit `__init_subclass__()` qui permet à une classe de participer à la création de sa sous-classe.

Examples

Métaclasses de base

Lorsque `type` est appelé avec trois arguments, il se comporte comme la classe (méta) et crée une nouvelle instance, c.-à-d. il produit une nouvelle classe / type.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__               # <type 'type'>
Dummy().__class__.__class__   # <type 'type'>
```

Il est possible de sous-classer le `type` pour créer une métaclasse personnalisée.

```
class mytype(type):
    def __init__(cls, name, bases, dict):
        # call the base initializer
        type.__init__(cls, name, bases, dict)

        # perform custom initialization...
        cls.__custom_attribute__ = 2
```

Maintenant, nous avons une nouvelle métaclasse personnalisée `mytype` qui peut être utilisée pour créer des classes de la même manière que le `type` .

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__               # <class '__main__.mytype'>
MyDummy().__class__.__class__   # <class '__main__.mytype'>
MyDummy.__custom_attribute__    # 2
```

Lorsque nous créons une nouvelle classe en utilisant le mot-clé `class` la métaclasse est choisie par défaut en fonction des classes de base.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

Dans l'exemple ci-dessus, la seule classe de base est `object` donc notre métaclasse sera le type d' `object` , qui est le `type` . Il est possible de remplacer la valeur par défaut, mais cela dépend si nous utilisons Python 2 ou Python 3:

Python 2.x 2.7

Un attribut spécial au niveau de la classe `__metaclass__` peut être utilisé pour spécifier la métaclasse.

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy)  # <class '__main__.mytype'>
```

Python 3.x 3.0

Un argument spécial de mot-clé de `metaclass` spécifie la métaclasse.

```
class MyDummy(metaclass=mytype):
    pass
type(MyDummy)  # <class '__main__.mytype'>
```

Tout argument de mot-clé (sauf la `metaclass`) dans la déclaration de classe sera transmis à la métaclasse. Ainsi, la `class MyDummy(metaclass=mytype, x=2)` transmettra `x=2` comme argument mot-clé au constructeur `mytype` .

Lisez cette description détaillée des méta-classes de python pour plus de détails.

Singletons utilisant des métaclasses

Un singleton est un modèle qui limite l'instanciation d'une classe à une instance / objet. Pour plus d'informations sur les modèles de conception de singleton python, voir ici .

```
class SingletonType(type):
    def __call__(cls, *args, **kwargs):
        try:
            return cls.__instance
        except AttributeError:
            cls.__instance = super(SingletonType, cls).__call__(*args, **kwargs)
            return cls.__instance
```

Python 2.x 2.7

```
class MySingleton(object):
    __metaclass__ = SingletonType
```

Python 3.x 3.0

```
class MySingleton(metaclass=SingletonType):
    pass
```

```
MySingleton() is MySingleton()  # True, only one instantiation occurs
```

Utiliser une métaclasse

La syntaxe de la métaclasse

Python 2.x 2.7

```
class MyClass(object):
    __metaclass__ = SomeMetaclass
```

Python 3.x 3.0

```
class MyClass(metaclass=SomeMetaclass):
    pass
```

Compatibilité Python 2 et 3 avec six

```
import six

class MyClass(six.with_metaclass(SomeMetaclass)):
    pass
```

Fonctionnalité personnalisée avec des métaclasses

La fonctionnalité dans les métaclasses peut être modifiée de sorte que chaque fois qu'une classe est construite, une chaîne est imprimée sur la sortie standard ou une exception est levée. Cette métaclasse affichera le nom de la classe en cours de construction.

```
class VerboseMetaclass(type):

    def __new__(cls, class_name, class_parents, class_dict):
        print("Creating class ", class_name)
        new_class = super().__new__(cls, class_name, class_parents, class_dict)
        return new_class
```

Vous pouvez utiliser la métaclasse comme ceci:

```
class Spam(metaclass=VerboseMetaclass):
```

```
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

La sortie standard sera:

```
Creating class Spam
[insert example string here]
```

Introduction aux Métaclasses

Qu'est-ce qu'une métaclasse?

En Python, tout est un objet: les entiers, les chaînes de caractères, les listes, même les fonctions et les classes elles-mêmes sont des objets. Et chaque objet est une instance d'une classe.

Pour vérifier la classe d'un objet x, on peut appeler le `type(x)` , donc:

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

La plupart des classes de python sont des instances de `type` . `type` lui-même est aussi une classe. Ces classes dont les instances sont aussi des classes sont appelées métaclasses.

La métaclasse la plus simple

OK, donc il y a déjà une métaclasse en Python: `type` . Pouvons-nous en créer un autre?

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

Cela n'ajoute aucune fonctionnalité, mais c'est une nouvelle métaclasse, voyez que MyClass est maintenant une instance de SimplestMetaclass:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

Une métaclasse qui fait quelque chose

Une méta - classe qui fait quelque chose remplace généralement le `type` de `__new__` , de modifier certaines propriétés de la classe à créer, avant d' appeler l'original `__new__` qui crée la classe:

```
class AnotherMetaclass(type):
    def __new__(cls, name, parents, dct):
        # cls is this class
        # name is the name of the class to be created
        # parents is the list of the class's parent classes
        # dct is the list of class's attributes (methods, static variables)

        # here all of the attributes can be modified before creating the class, e.g.

        dct['x'] = 8  # now the class will have a static variable x = 8

        # return value is the new class. super will take care of that
        return super(AnotherMetaclass, cls).__new__(cls, name, parents, dct)
```

La métaclasse par défaut

Vous avez peut-être entendu dire que tout en Python est un objet. C'est vrai, et tous les objets ont une classe:

```
>>> type(1)
int
```

Le littéral 1 est une instance de `int` . Permet de déclarer une classe:

```
>>> class Foo(object):
...     pass
...
```

Maintenant, permet de l'instancier:

```
>>> bar = Foo()
```

Quelle est la classe de `bar` ?

```
>>> type(bar)
Foo
```

Nice, le `bar` est une instance de `Foo` . Mais quelle est la classe de `Foo` elle-même?

```
>>> type(Foo)
type
```

Ok, `Foo` lui-même est une instance de `type` . Qu'en est-il du `type` lui-même?

```
>>> type(type)
```

```
type
```

Alors, quelle est une métaclasse? Pour l'instant, supposons que ce n'est qu'un nom de fantaisie pour la classe d'une classe. Plats à emporter:

- Tout est un objet en Python, donc tout a une classe
- La classe d'une classe s'appelle une métaclasse
- La métaclasse par défaut est `type` , et de loin c'est la métaclasse la plus courante

Mais pourquoi devriez-vous connaître les métaclasses? Eh bien, Python lui-même est assez "piratable", et le concept de métaclasse est important si vous faites des choses avancées comme la méta-programmation ou si vous voulez contrôler la façon dont vos classes sont initialisées.

Lire Métaclasses en ligne: https://riptutorial.com/fr/python/topic/286/metaclasses

Chapitre 114: Méthodes de chaîne

Syntaxe

- str.capitalize () -> str
- str.casefold () -> str [uniquement pour Python> 3.3]
- str.center (width [, fillchar]) -> str
- str.count (sub [, start [, end]]) -> int
- str.decode (encoding = "utf-8" [, des erreurs]) -> unicode [uniquement dans Python 2.x]
- str.encode (encoding = "utf-8", errors = "strict") -> octets
- str.endswith (suffixe [, début [, fin]]) -> bool
- str.expandtabs (tabsize = 8) -> str
- str.find (sub [, start [, end]]) -> int
- str.format (* args, ** kwargs) -> str
- str.format_map (mapping) -> str
- str.index (sub [, start [, end]]) -> int
- str.isalnum () -> bool
- str.isalpha () -> bool
- str.isdecimal () -> bool
- str.isdigit () -> bool
- str.isidentifier () -> bool
- str.islower () -> bool
- str.isnumeric () -> bool
- str.isprintable () -> bool
- str.isspace () -> bool
- str.istitle () -> bool
- str.isupper () -> bool
- str.join (itérable) -> str
- str.ljust (width [, fillchar]) -> str
- str.lower () -> str
- str.lstrip ([chars]) -> str
- static str.maketrans (x [, y [, z]])
- str.partition (sep) -> (tête, sep, queue)
- str.replace (old, new [, count]) -> str
- str.rfind (sub [, start [, end]]) -> int
- str.rindex (sub [, start [, end]]) -> int
- str.rjust (width [, fillchar]) -> str
- str.rpartition (sep) -> (tête, sep, queue)
- str.rsplit (sep = Aucun, maxsplit = -1) -> liste des chaînes
- str.rstrip ([chars]) -> str
- str.split (sep = Aucun, maxsplit = -1) -> liste de chaînes
- str.splitlines ([keepends]) -> liste des chaînes
- str.startswith (préfixe [, début [, fin]]) -> livre
- str.strip ([chars]) -> str

- str.swapcase () -> str
- str.title () -> str
- str.translate (table) -> str
- str.upper () -> str
- str.zfill (width) -> str

Remarques

Les objets String sont immuables, ce qui signifie qu'ils ne peuvent pas être modifiés de la même manière qu'une liste. De ce fait, les méthodes du type intégré `str` renvoient toujours un **nouvel** objet `str` , qui contient le résultat de l'appel de la méthode.

Examples

Changer la capitalisation d'une chaîne

Le type de chaîne de Python fournit de nombreuses fonctions qui agissent sur la capitalisation d'une chaîne. Ceux-ci inclus :

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

Avec les chaînes Unicode (la valeur par défaut dans Python 3), ces opérations **ne** sont **pas des correspondances** 1: 1 ou réversibles. La plupart de ces opérations sont destinées à l'affichage, plutôt qu'à la normalisation.

Python 3.x 3.3

`str.casefold()`

`str.casefold` crée une chaîne en minuscules qui convient aux comparaisons insensibles à la casse. Cette `str.lower` est plus agressive que `str.lower` et peut modifier des chaînes qui sont déjà en minuscules ou dont la longueur augmente, et qui ne sont pas destinées à l'affichage.

```
"XßΣ".casefold()
# 'xssσ'

"XßΣ".lower()
# 'xßς'
```

Les transformations qui ont lieu sous le regroupement sont définies par le consortium Unicode dans le fichier CaseFolding.txt sur leur site Web.

---

str.upper()

`str.upper` prend chaque caractère d'une chaîne et le convertit en son équivalent en majuscule, par exemple:

```
"This is a 'string'.".upper()
# "THIS IS A 'STRING'."
```

---

str.lower()

`str.lower` fait le contraire; il prend chaque caractère dans une chaîne et le convertit en son équivalent minuscule:

```
"This IS a 'string'.".lower()
# "this is a 'string'."
```

---

str.capitalize()

`str.capitalize` renvoie une version en majuscule de la chaîne, c'est-à-dire que le premier caractère est en majuscule et le reste plus bas:

```
"this Is A 'String'.".capitalize() # Capitalizes the first character and lowercases all others
# "This is a 'string'."
```

---

str.title()

`str.title` renvoie la version du titre de la chaîne, c'est-à-dire que chaque lettre au début d'un mot est en majuscule et que toutes les autres sont en minuscule:

```
"this Is a 'String'".title()
# "This Is A 'String'"
```

---

str.swapcase()

`str.swapcase` renvoie un nouvel objet chaîne dans lequel tous les caractères minuscules sont remplacés par des majuscules et tous les caractères majuscules vers le bas:

```
"this iS A STRiNG".swapcase() #Swaps case of each character
# "THIS Is a strIng"
```

Utilisation en tant que méthodes de classe `str`

Il convient de noter que ces méthodes peuvent être appelées soit sur des objets de type chaîne (voir ci-dessus), soit comme méthode de classe de la classe `str` (avec un appel explicite à `str.upper` , etc.).

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

Ceci est très utile lorsque vous appliquez l'une de ces méthodes à plusieurs chaînes en même temps, par exemple, une fonction de *map* .

```
map(str.upper,["These","are","some","'strings'"])
# ['THESE', 'ARE', 'SOME', "'STRINGS'"]
```

Diviser une chaîne basée sur un délimiteur en une liste de chaînes

str.split(sep=None, maxsplit=-1)

`str.split` prend une chaîne et retourne une liste de sous-chaînes de la chaîne d'origine. Le comportement diffère selon que l'argument `sep` est fourni ou omis.

Si `sep` n'est pas fourni, ou est `None` , alors le fractionnement a lieu partout où il y a des espaces. Toutefois, les espaces blancs de début et de fin sont ignorés et plusieurs caractères d'espacement consécutifs sont traités de la même manière qu'un seul caractère d'espacement:

```
>>> "This is a sentence.".split()
['This', 'is', 'a', 'sentence.']

>>> " This is    a sentence.  ".split()
['This', 'is', 'a', 'sentence.']

>>> "            ".split()
[]
```

Le paramètre `sep` peut être utilisé pour définir une chaîne de délimiteur. La chaîne d'origine est fractionnée à l'endroit où se trouve la chaîne de délimiteur et le délimiteur lui-même est supprimé. Plusieurs délimiteurs consécutifs *ne* sont *pas* traités comme une seule occurrence, mais créent plutôt des chaînes vides.

```
>>> "This is a sentence.".split(' ')
['This', 'is', 'a', 'sentence.']

>>> "Earth,Stars,Sun,Moon".split(',')
['Earth', 'Stars', 'Sun', 'Moon']

>>> " This is    a sentence.  ".split(' ')
['', 'This', 'is', '', '', '', 'a', 'sentence.', '', '']

>>> "This is a sentence.".split('e')
['This is a s', 'nt', 'nc', '.']

>>> "This is a sentence.".split('en')
['This is a s', 't', 'ce.']
```

La valeur par défaut consiste à diviser *chaque* occurrence du délimiteur, mais le paramètre `maxsplit` limite le nombre de fractionnements. La valeur par défaut de `-1` signifie aucune limite:

```
>>> "This is a sentence.".split('e', maxsplit=0)
['This is a sentence.']

>>> "This is a sentence.".split('e', maxsplit=1)
['This is a s', 'ntence.']

>>> "This is a sentence.".split('e', maxsplit=2)
['This is a s', 'nt', 'nce.']

>>> "This is a sentence.".split('e', maxsplit=-1)
['This is a s', 'nt', 'nc', '.']
```

`str.rsplit(sep=None, maxsplit=-1)`

`str.rsplit` ("split droit") diffère de `str.split` ("split gauche") lorsque `maxsplit` est spécifié. Le fractionnement commence à la fin de la chaîne plutôt qu'au début:

```
>>> "This is a sentence.".rsplit('e', maxsplit=1)
['This is a sentenc', '.']

>>> "This is a sentence.".rsplit('e', maxsplit=2)
['This is a sent', 'nc', '.']
```

Remarque : Python spécifie le nombre maximal de *divisions* effectuées, tandis que la plupart des autres langages de programmation spécifient le nombre maximal de *sous - chaînes* créées. Cela peut créer de la confusion lors du portage ou de la comparaison du code.

Remplacer toutes les occurrences d'une sous-chaîne par une autre sous-chaîne

Le type `str` de Python a également une méthode pour remplacer les occurrences d'une sous-chaîne par une autre sous-chaîne dans une chaîne donnée. Pour les cas plus exigeants, on peut utiliser `re.sub` .

`str.replace(old, new[, count])` :

`str.replace` prend deux arguments `old` et `new` contenant l' `old` sous-chaîne à remplacer par la `new` sous-chaîne. L'argument optionnel `count` spécifie le nombre de remplacements à effectuer:

Par exemple, pour remplacer `'foo'` par `'spam'` dans la chaîne suivante, on peut appeler `str.replace` avec `old = 'foo'` et `new = 'spam'` :

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
"Make sure to spam your sentence."
```

Si la chaîne donnée contient plusieurs exemples correspondant à l' `old` argument, **toutes les** occurrences sont remplacées par la valeur fournie dans `new` :

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
```

```
"It can spam multiple examples of spam if you want."
```

à moins, bien sûr, que nous fournissions une valeur pour le `count` . Dans ce cas, les occurrences de `count` vont être remplacées:

```
>>> """It can foo multiple examples of foo if you want, \
... or you can limit the foo with the third argument.""".replace('foo', 'spam', 1)
'It can spam multiple examples of foo if you want, or you can limit the foo with the third
argument.'
```

str.format et f-strings: mettre en forme les valeurs dans une chaîne

Python fournit des fonctionnalités d'interpolation et de formatage de chaînes via la fonction `str.format` , introduite dans la version 2.6 et des f-strings introduites dans la version 3.6.

Compte tenu des variables suivantes:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

Les énoncés suivants sont tous équivalents

```
"10 1.5 foo ['a', 1, 2] {'a': 1, 2: 'foo'}"
```

```
>>> "{} {} {} {} {}".format(i, f, s, l, d)

>>> str.format("{} {} {} {} {}", i, f, s, l, d)

>>> "{0} {1} {2} {3} {4}".format(i, f, s, l, d)

>>> "{0:d} {1:0.1f} {2} {3!r} {4!r}".format(i, f, s, l, d)

>>> "{i:d} {f:0.1f} {s} {l!r} {d!r}".format(i=i, f=f, s=s, l=l, d=d)
```

```
>>> f"{i} {f} {s} {l} {d}"

>>> f"{i:d} {f:0.1f} {s} {l!r} {d!r}"
```

Pour référence, Python prend également en charge les qualificateurs de style C pour le formatage des chaînes. Les exemples ci-dessous sont équivalents à ceux ci-dessus, mais les versions de `str.format` sont préférées en raison des avantages de la flexibilité, de la cohérence de la notation et de l'extensibilité:

```
"%d %0.1f %s %r %r" % (i, f, s, l, d)

"%(i)d %(f)0.1f %(s)s %(l)r %(d)r" % dict(i=i, f=f, s=s, l=l, d=d)
```

Les accolades `str.format` pour l'interpolation dans `str.format` peuvent également être numérotées

pour réduire la duplication lors du formatage des chaînes. Par exemple, les éléments suivants sont équivalents:

```
"I am from Australia. I love cupcakes from Australia!"
```

```
>>> "I am from {}. I love cupcakes from {}!".format("Australia", "Australia")

>>> "I am from {0}. I love cupcakes from {0}!".format("Australia")
```

Alors que la documentation officielle de python est, comme d'habitude, suffisamment complète, pyformat.info propose un grand nombre d'exemples avec des explications détaillées.

De plus, les caractères `{` et `}` peuvent être échappés en utilisant des crochets doubles:

```
"{'a': 5, 'b': 6}"
```

```
>>> "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)

>>> f"{{'{'a'}': {5}, '{'b'}': {6}}"
```

Voir Formatage de chaîne pour plus d'informations. `str.format()` été proposé dans PEP 3101 et les chaînes de caractères dans PEP 498 .

Comptage du nombre de fois qu'une sous-chaîne apparaît dans une chaîne

Une méthode est disponible pour compter le nombre d'occurrences d'une sous-chaîne dans une autre chaîne, `str.count` .

```
str.count(sub[, start[, end]])
```

`str.count` renvoie un `int` indiquant le nombre d'occurrences de non-chevauchement du sous-chaîne `sub` dans une autre chaîne. Les arguments optionnels `start` et `end` indiquent le début et la fin de la recherche. Par défaut `start = 0` et `end = len(str)` signifiant que la chaîne entière sera recherchée:

```
>>> s = "She sells seashells by the seashore."
>>> s.count("sh")
2
>>> s.count("se")
3
>>> s.count("sea")
2
>>> s.count("seashells")
1
```

En spécifiant une valeur différente pour `start` , `end` , nous pouvons obtenir une recherche plus localisée et compte, par exemple, si `start` est égal à `13` l'appel à:

```
>>> s.count("sea", start)
```

```
1
```

est équivalent à:

```
>>> t = s[start:]
>>> t.count("sea")
1
```

Tester les caractères de début et de fin d'une chaîne

Pour tester le début et la fin d'une chaîne donnée en Python, on peut utiliser les méthodes `str.startswith()` et `str.endswith()` .

`str.startswith(prefix[, start[, end]])`

Comme son nom l'indique, `str.startswith` est utilisé pour tester si une chaîne donnée commence par les caractères donnés dans le `prefix` .

```
>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False
```

Les arguments optionnels `start` et `end` spécifient les points de début et de fin à partir desquels les tests vont commencer et se terminer. Dans l'exemple suivant, en spécifiant une valeur de départ de `2` notre chaîne sera recherchée à partir de la position `2` et après:

```
>>> s.startswith("is", 2)
True
```

Cela donne `True` puisque `s[2] == 'i'` et `s[3] == 's'` .

Vous pouvez également utiliser un `tuple` pour vérifier s'il commence par un jeu de chaînes

```
>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False
```

`str.endswith(prefix[, start[, end]])`

`str.endswith` est exactement similaire à `str.startswith` la seule différence étant qu'il recherche les caractères de fin et non les caractères de départ. Par exemple, pour tester si une chaîne se termine par un arrêt complet, on pourrait écrire:

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

comme avec les `startswith` plus d'un caractère peut être utilisé comme séquence de fin:

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

Vous pouvez également utiliser un `tuple` pour vérifier s'il se termine par un jeu de chaînes

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

Test de la composition d'une chaîne

Le type `str` de Python comporte également un certain nombre de méthodes permettant d'évaluer le contenu d'une chaîne. Ce sont `str.isalpha` , `str.isdigit` , `str.isalnum` , `str.isspace` . La capitalisation peut être testée avec `str.isupper` , `str.islower` et `str.istitle` .

`str.isalpha`

`str.isalpha` ne prend aucun argument et renvoie `True` si tous les caractères d'une chaîne donnée sont alphabétiques, par exemple:

```
>>> "Hello World".isalpha()  # contains a space
False
>>> "Hello2World".isalpha()  # contains a number
False
>>> "HelloWorld!".isalpha()  # contains punctuation
False
>>> "HelloWorld".isalpha()
True
```

En tant que casse, la chaîne vide est évaluée à `False` lorsqu'elle est utilisée avec `"".isalpha()` .

`str.isupper` , `str.islower` , `str.istitle`

Ces méthodes testent la capitalisation dans une chaîne donnée.

`str.isupper` est une méthode qui renvoie `True` si tous les caractères d'une chaîne donnée sont en majuscule et `False` sinon.

```
>>> "HeLLO WORLD".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> "".isupper()
False
```

Inversement, `str.islower` est une méthode qui renvoie `True` si tous les caractères d'une chaîne donnée sont en minuscule et `False` .

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` retourne `True` si la chaîne donnée est `str.istitle` le titre; c'est-à-dire que chaque mot commence par un caractère majuscule suivi de caractères minuscules.

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

`str.isdecimal` , `str.isdigit` , `str.isnumeric`

`str.isdecimal` renvoie si la chaîne est une séquence de chiffres décimaux, appropriée pour représenter un nombre décimal.

`str.isdigit` inclut des chiffres qui ne sont pas sous une forme appropriée pour représenter un nombre décimal, tel que des chiffres en exposant.

`str.isnumeric` inclut toutes les valeurs numériques, même si ce ne sont pas des chiffres, telles que des valeurs en dehors de l'intervalle 0-9.

```
            isdecimal    isdigit   isnumeric

12345        True         True       True
□2□□5        True         True       True
①²³□₅       False       True       True
⑩□           False       False      True
Five         False       False      False
```

Les bytestrings (`bytes` dans Python 3, `str` dans Python 2), ne supportent que `isdigit` , qui vérifie uniquement les chiffres ASCII de base.

Comme avec `str.isalpha` , la chaîne vide a la valeur `False` .

---

str.isalnum

Il s'agit d'une combinaison de `str.isalpha` et de `str.isnumeric` , en particulier de `True` si tous les caractères de la chaîne donnée sont **alphanumériques** , c'est-à-dire qu'ils sont composés de caractères alphabétiques *ou* numériques:

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
>>> "Hello World".isalnum()  # contains whitespace
False
```

---

str.isspace

Évalue à `True` si la chaîne ne contient que des caractères d'espacement.

```
>>> "\t\r\n".isspace()
True
>>> " ".isspace()
True
```

Parfois, une chaîne a l'air “vide” mais nous ne savons pas si c'est parce qu'elle contient uniquement des espaces ou pas de caractères du tout.

```
>>> "".isspace()
False
```

Pour couvrir ce cas, nous avons besoin d'un test supplémentaire

```
>>> my_str = ''
>>> my_str.isspace()
False
>>> my_str.isspace() or not my_str
True
```

Mais le moyen le plus court de tester si une chaîne est vide ou contient simplement des caractères d'espacement est d'utiliser `strip` (sans arguments, cela supprime tous les caractères blancs avant et après).

```
>>> not my_str.strip()
True
```

str.translate: Traduction de caractères dans une chaîne

Python prend en charge une méthode de `translate` sur le type `str` qui vous permet de spécifier la table de traduction (utilisée pour les remplacements) ainsi que tous les caractères à supprimer dans le processus.

```
str.translate(table[, deletechars])
```

Paramètre	La description
`table`	C'est une table de correspondance qui définit le mappage d'un caractère à un autre.
`deletechars`	Une liste de caractères à supprimer de la chaîne.

La méthode `maketrans` (`str.maketrans` dans Python 3 et `string.maketrans` dans Python 2) vous permet de générer une table de traduction.

```
>>> translation_table = str.maketrans("aeiou", "12345")
>>> my_string = "This is a string!"
>>> translated = my_string.translate(translation_table)
'Th3s 3s 1 str3ng!'
```

La méthode `translate` renvoie une chaîne qui est une copie traduite de la chaîne d'origine.

Vous pouvez définir l'argument de la `table` sur `None` si vous devez uniquement supprimer des caractères.

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
'ths syntx s vry sfl'
```

Retirer des caractères de début / fin indésirables d'une chaîne

Trois méthodes permettent de `str.strip` les caractères de `str.strip` et de fin d'une chaîne: `str.strip` , `str.rstrip` et `str.lstrip` . Les trois méthodes ont la même signature et toutes les trois renvoient un nouvel objet chaîne avec des caractères indésirables supprimés.

```
str.strip([chars])
```

`str.strip` agit sur une chaîne donnée et supprime (strip) les caractères de `str.strip` ou de fin contenus dans les `chars` l'argument; Si les `chars` ne sont pas fournis ou sont `None` , tous les espaces sont supprimés par défaut. Par exemple:

```
>>> "    a line with leading and trailing space     ".strip()
'a line with leading and trailing space'
```

Si des `chars` sont fournis, tous les caractères qu'il contient sont supprimés de la chaîne, qui est renvoyée. Par exemple:

```
>>> ">>> a Python prompt".strip('> ')  # strips '>' character and space character
'a Python prompt'
```

`str.rstrip([chars])` et `str.lstrip([chars])`

Ces méthodes ont une sémantique et des arguments similaires avec `str.strip()` , leur différence se situe dans la direction de départ. `str.rstrip()` commence à la fin de la chaîne alors que `str.lstrip()` se sépare du début de la chaîne.

Par exemple, en utilisant `str.rstrip` :

```
>>> "     spacious string      ".rstrip()
'     spacious string'
```

En utilisant `str.lstrip` :

```
>>> "     spacious string      ".rstrip()
'spacious string      '
```

Comparaisons de chaînes insensibles à la casse

Comparer une chaîne de manière insensible à la casse semble être quelque chose d'anodin, mais ce n'est pas le cas. Cette section ne considère que les chaînes Unicode (la valeur par défaut dans Python 3). Notez que Python 2 peut avoir des faiblesses subtiles par rapport à Python 3 - la gestion unicode ultérieure est beaucoup plus complète.

La première chose à noter est que les conversions de suppression de casse dans Unicode ne sont pas triviales. Il y a du texte pour `text.lower() != text.upper().lower()` , tel que `"ß"` :

```
>>> "ß".lower()
'ß'

>>> "ß".upper().lower()
'ss'
```

Mais disons que vous vouliez comparer sans `"BUSSE"` et `"Buße"` . Heck, vous voulez probablement aussi comparer `"BUSSE"` et `"BU□E"` égaux - c'est la nouvelle forme de capital. La méthode recommandée est d'utiliser le `casefold` :

Python 3.x 3.3

```
>>> help(str.casefold)
"""
Help on method_descriptor:

casefold(...)
      S.casefold() -> str

     Return a version of S suitable for caseless comparisons.
"""
```

Ne pas utiliser `lower` . Si le `casefold` n'est pas disponible, faire `.upper().lower()` aide (mais seulement un peu).

Ensuite, vous devriez considérer les accents. Si votre moteur de rendu est bon, vous pensez probablement `"ê" == "ê "` - mais ce n'est pas le cas:

```
>>> "ê" == "ê "
False
```

C'est parce qu'ils sont en fait

```
>>> import unicodedata

>>> [unicodedata.name(char) for char in "ê"]
['LATIN SMALL LETTER E WITH CIRCUMFLEX']

>>> [unicodedata.name(char) for char in "ê "]
['LATIN SMALL LETTER E', 'COMBINING CIRCUMFLEX ACCENT']
```

La manière la plus simple de gérer cela est `unicodedata.normalize` . Vous souhaitez probablement utiliser la normalisation **NFKD** , mais n'hésitez pas à consulter la documentation. Alors on fait

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê ")
True
```

Pour finir, ceci est exprimé en fonctions:

```
import unicodedata

def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

Joindre une liste de chaînes dans une chaîne

Une chaîne peut être utilisée comme séparateur pour joindre une liste de chaînes en une seule chaîne à l'aide de la méthode `join()` . Par exemple, vous pouvez créer une chaîne où chaque élément d'une liste est séparé par un espace.

```
>>> " ".join(["once","upon","a","time"])
"once upon a time"
```

L'exemple suivant sépare les éléments de chaîne avec trois traits d'union.

```
>>> "---".join(["once", "upon", "a", "time"])
"once---upon---a---time"
```

Les constantes utiles du module String

Le module de `string` de Python fournit des constantes pour les opérations liées aux chaînes. Pour les utiliser, importez le module de `string` :

```
>>> import string
```

`string.ascii_letters` :

Concaténation de `ascii_lowercase` et `ascii_uppercase` :

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.ascii_lowercase` :

Contient tous les caractères ASCII minuscules:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

`string.ascii_uppercase` :

Contient tous les caractères ASCII majuscules:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.digits` :

Contient tous les caractères numériques décimaux:

```
>>> string.digits
'0123456789'
```

`string.hexdigits` :

Contient tous les caractères hexadécimaux:

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

`string.octaldigits` :

Contient tous les caractères numériques octaux:

```
>>> string.octaldigits
'01234567'
```

`string.punctuation` :

Contient tous les caractères considérés comme des signes de ponctuation dans les paramètres régionaux `C` :

```
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

`string.whitespace` :

Contient tous les caractères ASCII considérés comme des espaces:

```
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

En mode script, `print(string.whitespace)` imprimera les caractères réels, utilisez `str` pour obtenir la chaîne retournée ci-dessus.

`string.printable` :

Contient tous les caractères considérés imprimables. une combinaison de `string.digits` , `string.ascii_letters` , `string.punctuation` et `string.whitespace` .

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-
./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Inverser une chaîne

Une chaîne peut être inversée à l'aide de la fonction Reverse `reversed()` intégrée, qui prend une chaîne et renvoie un itérateur dans l'ordre inverse.

```
>>> reversed('hello')
<reversed object at 0x0000000000000000>
>>> [char for char in reversed('hello')]
['o', 'l', 'l', 'e', 'h']
```

`reversed()` peut être inclus dans un appel à `''.join()` pour créer une chaîne à partir de l'itérateur.

```
>>> ''.join(reversed('hello'))
'olleh'
```

Bien que l'utilisation de `reversed()` soit plus lisible pour les utilisateurs non initiés de Python, l'utilisation du découpage étendu avec un pas de `-1` est plus rapide et plus concise. Ici, essayez de le mettre en œuvre en tant que fonction:

```
>>> def reversed_string(main_string):
...     return main_string[::-1]
...
>>> reversed_string('hello')
'olleh'
```

Justifier les chaînes

Python fournit des fonctions pour justifier les chaînes, permettant un remplissage du texte pour faciliter l'alignement des différentes chaînes.

Voici un exemple de `str.ljust` et de `str.rjust` :

```
interstates_lengths = {
    5: (1381, 2222),
    19: (63, 102),
    40: (2555, 4112),
    93: (189,305),
}
for road, length in interstates_lengths.items():
    miles,kms = length
    print('{} -> {} mi. ({} km.)'.format(str(road).rjust(4), str(miles).ljust(4),
str(kms).ljust(4)))
```

```
  40 -> 2555 mi. (4112 km.)
  19 -> 63   mi. (102  km.)
   5 -> 1381 mi. (2222 km.)
  93 -> 189  mi. (305  km.)
```

`ljust` et `rjust` sont très similaires. Les deux ont un paramètre `width` et un paramètre facultatif `fillchar` . Toute chaîne créée par ces fonctions est au moins aussi longue que le paramètre `width` qui a été transmis à la fonction. Si la chaîne est plus longue que la `width` , elle n'est pas tronquée. L'argument `fillchar` , qui `fillchar` défaut le caractère d'espace `' '` doit être un caractère unique et non une chaîne à caractères multiples.

La `ljust` fonction Tapis de la fin de la chaîne , il est appelé avec la `fillchar` jusqu'à ce qu'il soit de `width` caractères. La fonction `rjust` le début de la chaîne de la même manière. Par conséquent, le `l` et le `r` dans les noms de ces fonctions font référence au côté dans lequel la chaîne d'origine, *pas le* *`fillchar`* , est positionnée dans la chaîne de sortie.

Conversion entre les données str ou bytes et les caractères unicode

Le contenu des fichiers et des messages réseau peut représenter des caractères codés. Ils doivent souvent être convertis en Unicode pour un affichage correct.

Dans Python 2, vous devrez peut-être convertir des données str en caractères Unicode. La valeur par défaut (`''` , `""` , etc.) est une chaîne ASCII, les valeurs en dehors de la plage ASCII étant affichées en tant que valeurs échappées. Les chaînes Unicode sont `u''` (ou `u""` , etc.).

Python 2.x 2.3

```
# You get "© abc" encoded in UTF-8 from a file, network, or other data source

s = '\xc2\xa9 abc'  # s is a byte array, not a string of characters
                    # Doesn't know the original was UTF-8
                    # Default form of string literals in Python 2
s[0]                # '\xc2' - meaningless byte (without context such as an encoding)
type(s)             # str - even though it's not a useful one w/o having a known encoding

u = s.decode('utf-8')  # u'\xa9 abc'
                       # Now we have a Unicode string, which can be read as UTF-8 and printed
properly
                       # In Python 2, Unicode string literals need a leading u
                       # str.decode converts a string which may contain escaped bytes to a
Unicode string
u[0]                # u'\xa9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # unicode

u.encode('utf-8')   # '\xc2\xa9 abc'
                    # unicode.encode produces a string with escaped bytes for non-ASCII
characters
```

Dans Python 3, vous devrez peut-être convertir des tableaux d'octets (appelés «littéraux d'octets») en chaînes de caractères Unicode. La valeur par défaut est désormais une chaîne Unicode, et ByteString littéraux doit maintenant être entré comme `b''` , `b""` , etc. Un octet littéral retourne `True` à `isinstance(some_val, byte)` , en supposant `some_val` être une chaîne qui pourrait être codé en octets.

Python 3.x 3.0

```
# You get from file or network "© abc" encoded in UTF-8

s = b'\xc2\xa9 abc' # s is a byte array, not characters
                    # In Python 3, the default string literal is Unicode; byte array literals
need a leading b
s[0]                # b'\xc2' - meaningless byte (without context such as an encoding)
type(s)             # bytes - now that byte arrays are explicit, Python can show that.

u = s.decode('utf-8')  # '© abc' on a Unicode terminal
                       # bytes.decode converts a byte array to a string (which will, in Python
3, be Unicode)
u[0]                # '\u00a9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)             # str
                    # The default string literal in Python 3 is UTF-8 Unicode

u.encode('utf-8')   # b'\xc2\xa9 abc'
                    # str.encode produces a byte array, showing ASCII-range bytes as unescaped
characters.
```

Chaîne contient

Python le rend extrêmement intuitif pour vérifier si une chaîne contient une sous-chaîne donnée. Utilisez simplement l'opérateur `in` :

```
>>> "foo" in "foo.baz.bar"
True
```

Note: tester une chaîne vide donnera toujours `True` :

```
>>> "" in "test"
True
```

Lire Méthodes de chaîne en ligne: https://riptutorial.com/fr/python/topic/278/methodes-de-chaine

Chapitre 115: Méthodes définies par l'utilisateur

Examples

Création d'objets de méthode définis par l'utilisateur

Des objets de méthode définis par l'utilisateur peuvent être créés lors de l'obtention d'un attribut de classe (peut-être via une instance de cette classe), si cet attribut est un objet fonction défini par l'utilisateur, un objet de méthode défini par l'utilisateur.

```
class A(object):
    # func: A user-defined function object
    #
    # Note that func is a function object when it's defined,
    # and an unbound method object when it's retrieved.
    def func(self):
        pass

    # classMethod: A class method
    @classmethod
    def classMethod(self):
        pass

class B(object):
    # unboundMeth: A unbound user-defined method object
    #
    # Parent.func is an unbound user-defined method object here,
    # because it's retrieved.
    unboundMeth = A.func

a = A()
b = B()

print A.func
# output: <unbound method A.func>
print a.func
# output: <bound method A.func of <__main__.A object at 0x10e9ab910>>
print B.unboundMeth
# output: <unbound method A.func>
print b.unboundMeth
# output: <unbound method A.func>
print A.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
print a.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
```

Lorsque l'attribut est un objet de méthode défini par l'utilisateur, un nouvel objet de méthode n'est créé que si la classe à partir de laquelle il est extrait est identique à la classe stockée dans l'objet de méthode d'origine ou à une classe dérivée de celle-ci; sinon, l'objet méthode d'origine est utilisé tel quel.

```
# Parent: The class stored in the original method object
class Parent(object):
    # func: The underlying function of original method object
    def func(self):
        pass
    func2 = func

# Child: A derived class of Parent
class Child(Parent):
    func = Parent.func

# AnotherClass: A different class, neither subclasses nor subclassed
class AnotherClass(object):
    func = Parent.func

print Parent.func is Parent.func                # False, new object created
print Parent.func2 is Parent.func2              # False, new object created
print Child.func is Child.func                  # False, new object created
print AnotherClass.func is AnotherClass.func    # True, original object used
```

Exemple de tortue

Voici un exemple d'utilisation d'une fonction définie par l'utilisateur à appeler plusieurs fois (multiple) dans un script.

```
import turtle, time, random #tell python we need 3 different modules
turtle.speed(0) #set draw speed to the fastest
turtle.colormode(255) #special colormode
turtle.pensize(4) #size of the lines that will be drawn
def triangle(size): #This is our own function, in the parenthesis is a variable we have
defined that will be used in THIS FUNCTION ONLY. This fucntion creates a right triangle
    turtle.forward(size) #to begin this function we go forward, the amount to go forward by is
the variable size
    turtle.right(90) #turn right by 90 degree
    turtle.forward(size) #go forward, again with variable
    turtle.right(135) #turn right again
    turtle.forward(size * 1.5) #close the triangle. thanks to the Pythagorean theorem we know
that this line must be 1.5 times longer than the other two(if they are equal)
while(1): #INFINITE LOOP
    turtle.setpos(random.randint(-200, 200), random.randint(-200, 200)) #set the draw point to
a random (x,y) position
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize the RGB color
    triangle(random.randint(5, 55)) #use our function, because it has only one variable we can
simply put a value in the parenthesis. The value that will be sent will be random between 5 -
55, end the end it really just changes ow big the triangle is.
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize color again
```

Lire Méthodes définies par l'utilisateur en ligne:
https://riptutorial.com/fr/python/topic/3965/methodes-definies-par-l-utilisateur

Chapitre 116: Mixins

Syntaxe

- class *ClassName* (*MainClass* , *Mixin1* , *Mixin2* , ...): # Utilisé pour déclarer une classe *nommée ClassName* , principale (première) classe *MainClass* et mixins *Mixin1* , *Mixin2* , etc.
- class *ClassName* (*Mixin1* , *MainClass* , *Mixin2* , ...): # La classe 'main' ne doit pas nécessairement être la première classe; il n'y a vraiment pas de différence entre elle et le mixin

Remarques

Ajouter un mixin à une classe ressemble beaucoup à l'ajout d'une superclasse, car c'est à peu près tout. Un objet d'une classe avec le mixin *Foo* sera également une instance de *Foo* , et `isinstance(instance, Foo)` retournera true

Examples

Mixin

Un **Mixin** est un ensemble de propriétés et de méthodes pouvant être utilisées dans différentes classes, qui *ne* proviennent *pas* d'une classe de base. En langage de programmation orienté objet, vous utilisez généralement l' *héritage* pour donner aux objets de classes différentes la même fonctionnalité; Si un ensemble d'objets possède une certaine capacité, vous mettez cette capacité dans une classe de base dont les deux objets *héritent* .

> Par exemple, disons que vous avez les classes `Car` , `Boat` et `Plane` . Les objets de toutes ces classes ont la capacité de voyager, ils ont donc la fonction de `travel` . Dans ce scénario, ils voyagent tous de la même manière de base; en empruntant un itinéraire et en le suivant. Pour implémenter cette fonction, vous pouvez dériver toutes les classes de `Vehicle` et placer la fonction dans cette classe partagée:

```
class Vehicle(object):
   """A generic vehicle class."""

   def __init__(self, position):
       self.position = position

   def travel(self, destination):
       route = calculate_route(from=self.position, to=destination)
       self.move_along(route)

class Car(Vehicle):
   ...

class Boat(Vehicle):
   ...
```

```
class Plane(Vehicle):
   ...
```

Avec ce code, vous pouvez appeler les `travel` en voiture (`car.travel("Montana")`), en bateau (`boat.travel("Hawaii")`) et en avion (`plane.travel("France")`)

Cependant, que se passe-t-il si vous avez des fonctionnalités qui ne sont pas disponibles pour une classe de base? Disons, par exemple, que vous voulez donner à `Car` une radio et la possibilité de l'utiliser pour jouer une chanson sur une station de radio, avec `play_song_on_station` , mais vous avez aussi une `Clock` qui peut aussi utiliser une radio. `Car` et `Clock` peuvent partager une classe de base (`Machine`). Cependant, toutes les machines ne peuvent pas jouer de chansons. `Boat` et `Plane` ne peuvent pas (au moins dans cet exemple). Alors, comment accomplissez-vous sans dupliquer le code? Vous pouvez utiliser un mixin. En Python, donner un mixin à une classe est aussi simple que de l'ajouter à la liste des sous-classes, comme ceci

```
class Foo(main_super, mixin): ...
```

`Foo` héritera de toutes les propriétés et méthodes de `main_super` , mais aussi de celles de `mixin` .

Alors, pour donner les classes de `Car` et l' horloge la possibilité d'utiliser une radio, vous pouvez passer outre `Car` du dernier exemple et écrire ceci:

```
class RadioUserMixin(object):
   def __init__(self):
       self.radio = Radio()

   def play_song_on_station(self, station):
       self.radio.set_station(station)
       self.radio.play_song()

class Car(Vehicle, RadioUserMixin):
   ...

class Clock(Vehicle, RadioUserMixin):
   ...
```

Maintenant, vous pouvez appeler `car.play_song_on_station(98.7)` et `clock.play_song_on_station(101.3)` , mais pas quelque chose comme `boat.play_song_on_station(100.5)`

L'important avec les mixins est qu'ils vous permettent d'ajouter des fonctionnalités à des objets très différents, qui ne partagent pas une sous-classe "principale" avec cette fonctionnalité, mais partagent néanmoins le code. Sans les mixins, faire quelque chose comme l'exemple ci-dessus serait beaucoup plus difficile et / ou pourrait nécessiter des répétitions.

Méthodes de substitution dans les mixins

Les mixins sont une sorte de classe utilisée pour "mélanger" des propriétés et des méthodes supplémentaires dans une classe. C'est généralement bien car plusieurs fois les classes mixin ne remplacent pas les méthodes des classes de base. Mais si vous remplacez des méthodes ou des

propriétés dans vos mixins, cela peut entraîner des résultats inattendus car en Python, la hiérarchie des classes est définie de droite à gauche.

Par exemple, prenez les cours suivants

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class BaseClass(object):
    def test(self):
        print "Base"

class MyClass(BaseClass, Mixin1, Mixin2):
    pass
```

Dans ce cas, la classe Mixin2 est la classe de base, étendue par Mixin1 et enfin par BaseClass. Ainsi, si nous exécutons l'extrait de code suivant:

```
>>> x = MyClass()
>>> x.test()
Base
```

Nous voyons que le résultat renvoyé provient de la classe de base. Cela peut conduire à des erreurs inattendues dans la logique de votre code et doit être pris en compte et gardé à l'esprit

Lire Mixins en ligne: https://riptutorial.com/fr/python/topic/4359/mixins

Chapitre 117: Modèles de conception

Introduction

Un modèle de conception est une solution générale à un problème courant dans le développement de logiciels. Cette rubrique de documentation vise spécifiquement à fournir des exemples de modèles de conception courants en Python.

Examples

Modèle de stratégie

Ce modèle de conception est appelé modèle de stratégie. Il est utilisé pour définir une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Le modèle de conception de stratégie permet à un algorithme de varier indépendamment des clients qui l'utilisent.

Par exemple, les animaux peuvent «marcher» de différentes manières. La marche pourrait être considérée comme une stratégie mise en œuvre par différents types d'animaux:

```
from types import MethodType


class Animal(object):

    def __init__(self, *args, **kwargs):
        self.name = kwargs.pop('name', None) or 'Animal'
        if kwargs.get('walk', None):
            self.walk = MethodType(kwargs.pop('walk'), self)

    def walk(self):
        """
        Cause animal instance to walk

        Walking funcionallity is a strategy, and is intended to
        be implemented separately by different types of animals.
        """
        message = '{} should implement a walk method'.format(
            self.__class__.__name__)
        raise NotImplementedError(message)


# Here are some different walking algorithms that can be used with Animal
def snake_walk(self):
    print('I am slithering side to side because I am a {}.'.format(self.name))

def four_legged_animal_walk(self):
    print('I am using all four of my legs to walk because I am a(n) {}.'.format(
        self.name))

def two_legged_animal_walk(self):
    print('I am standing up on my two legs to walk because I am a {}.'.format(
        self.name))
```

L'exécution de cet exemple produirait la sortie suivante:

```
generic_animal = Animal()
king_cobra = Animal(name='King Cobra', walk=snake_walk)
elephant = Animal(name='Elephant', walk=four_legged_animal_walk)
kangaroo = Animal(name='Kangaroo', walk=two_legged_animal_walk)

kangaroo.walk()
elephant.walk()
king_cobra.walk()
# This one will Raise a NotImplementedError to let the programmer
# know that the walk method is intended to be used as a strategy.
generic_animal.walk()

    # OUTPUT:
    #
    # I am standing up on my two legs to walk because I am a Kangaroo.
    # I am using all four of my legs to walk because I am a(n) Elephant.
    # I am slithering side to side because I am a King Cobra.
    # Traceback (most recent call last):
    #   File "./strategy.py", line 56, in <module>
    #     generic_animal.walk()
    #   File "./strategy.py", line 30, in walk
    #     raise NotImplementedError(message)
    # NotImplementedError: Animal should implement a walk method
```

Notez que dans des langages comme C ++ ou Java, ce modèle est implémenté en utilisant une classe abstraite ou une interface pour définir une stratégie. En Python, il est plus logique de définir des fonctions externes pouvant être ajoutées dynamiquement à une classe à l'aide de `types.MethodType` .

Introduction aux motifs de conception et Singleton Pattern

Les modèles de conception apportent des solutions aux `commonly occurring problems` de la conception de logiciels. Les modèles de conception ont été introduits pour la première fois par le `GoF(Gang of Four)` où ils ont décrit les modèles communs comme des problèmes qui se répètent et des solutions à ces problèmes.

Les modèles de conception ont quatre éléments essentiels:

1. `The pattern name` est un descripteur que nous pouvons utiliser pour décrire un problème de conception, ses solutions et ses conséquences en un mot ou deux.
2. `The problem` décrit quand appliquer le modèle.
3. `The solution` décrit les éléments qui composent la conception, leurs relations, leurs responsabilités et leurs collaborations.
4. `The consequences` sont les résultats et les compromis de l'application du modèle.

Avantages des modèles de conception:

1. Ils sont réutilisables dans plusieurs projets.
2. Le niveau architectural des problèmes peut être résolu
3. Ils ont fait leurs preuves et ont fait leurs preuves, ce qui est l'expérience des développeurs et des architectes.

4. Ils ont la fiabilité et la dépendance

Les modèles de conception peuvent être classés en trois catégories:

1. Motif Créatif
2. Motif structurel
3. Motif comportemental

`Creational Pattern` - Ils sont concernés par la façon dont l'objet peut être créé et ils isolent les détails de la création d'objet.

`Structural Pattern` - Ils conçoivent la structure des classes et des objets afin qu'ils puissent composer pour obtenir des résultats plus importants.

`Behavioral Pattern` - Ils concernent l'interaction entre les objets et la responsabilité des objets.

Motif Singleton :

C'est un type de `creational pattern` qui fournit un mécanisme pour n'avoir qu'un objet et un objet d'un type donné et fournit un point d'accès global.

Par exemple, Singleton peut être utilisé dans des opérations de base de données, où nous voulons que l'objet de base de données conserve la cohérence des données.

la mise en oeuvre

Nous pouvons implémenter Singleton Pattern dans Python en créant une seule instance de classe Singleton et en servant à nouveau le même objet.

```
class Singleton(object):
    def __new__(cls):
        # hasattr method checks if the class object an instance property or not.
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

s = Singleton()
print ("Object created", s)

s1 = Singleton()
print ("Object2 created", s1)
```

Sortie:

```
('Object created', <__main__.Singleton object at 0x10a7cc310>)
('Object2 created', <__main__.Singleton object at 0x10a7cc310>)
```

Notez que dans des langages comme C ++ ou Java, ce modèle est implémenté en rendant le constructeur privé et en créant une méthode statique qui effectue l'initialisation de l'objet. De cette façon, un objet est créé sur le premier appel et la classe retourne le même objet par la suite. Mais en Python, nous n'avons aucun moyen de créer des constructeurs privés.

Modèle d'usine

Le motif d'usine est également un `Creational pattern` . Le terme `factory` signifie qu'une classe est responsable de la création d'objets d'autres types. Il existe une classe qui agit en tant que fabrique et qui est associée à des objets et à des méthodes. Le client crée un objet en appelant les méthodes avec certains paramètres et fabrique l'objet du type souhaité et le renvoie au client.

```
from abc import ABCMeta, abstractmethod

class Music():
    __metaclass__ = ABCMeta
    @abstractmethod
    def do_play(self):
        pass

class Mp3(Music):
    def do_play(self):
        print ("Playing .mp3 music!")

class Ogg(Music):
    def do_play(self):
        print ("Playing .ogg music!")

class MusicFactory(object):
    def play_sound(self, object_type):
        return eval(object_type)().do_play()

if __name__ == "__main__":
    mf = MusicFactory()
    music = input("Which music you want to play Mp3 or Ogg")
    mf.play_sound(music)
```

Sortie:

```
Which music you want to play Mp3 or Ogg"Ogg"
Playing .ogg music!
```

`MusicFactory` est la classe de fabrique ici qui crée un objet de type `Mp3` ou `Ogg` fonction du choix de l'utilisateur.

Procuration

L'objet proxy est souvent utilisé pour garantir un accès sécurisé à un autre objet, quelle logique métier interne nous ne voulons pas polluer avec les exigences de sécurité.

Supposons que nous souhaitons garantir que seul l'utilisateur des autorisations spécifiques puisse accéder aux ressources.

Définition du proxy: (il garantit que seuls les utilisateurs capables de voir les réservations pourront utiliser le service reservation_service)

```
from datetime import date
from operator import attrgetter
```

```
class Proxy:
    def __init__(self, current_user, reservation_service):
        self.current_user = current_user
        self.reservation_service = reservation_service

    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        if self.current_user.can_see_reservations:
            return self.reservation_service.highest_total_price_reservations(
                date_from,
                date_to,
                reservations_count
              )
        else:
            return []

#Models and ReservationService:

class Reservation:
    def __init__(self, date, total_price):
        self.date = date
        self.total_price = total_price

class ReservationService:
    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        # normally it would be read from database/external service
        reservations = [
            Reservation(date(2014, 5, 15), 100),
            Reservation(date(2017, 5, 15), 10),
            Reservation(date(2017, 1, 15), 50)
        ]

        filtered_reservations = [r for r in reservations if (date_from <= r.date <= date_to)]

        sorted_reservations = sorted(filtered_reservations, key=attrgetter('total_price'),
reverse=True)

        return sorted_reservations[0:reservations_count]


class User:
    def __init__(self, can_see_reservations, name):
        self.can_see_reservations = can_see_reservations
        self.name = name

#Consumer service:

class StatsService:
    def __init__(self, reservation_service):
        self.reservation_service = reservation_service

    def year_top_100_reservations_average_total_price(self, year):
        reservations = self.reservation_service.highest_total_price_reservations(
            date(year, 1, 1),
            date(year, 12, 31),
            1
        )

        if len(reservations) > 0:
            total = sum(r.total_price for r in reservations)
```

```
            return total / len(reservations)
        else:
            return 0

#Test:
def test(user, year):
    reservations_service = Proxy(user, ReservationService())
    stats_service = StatsService(reservations_service)
    average_price = stats_service.year_top_100_reservations_average_total_price(year)
    print("{0} will see: {1}".format(user.name, average_price))

test(User(True, "John the Admin"), 2017)
test(User(False, "Guest"),         2017)
```

AVANTAGES

- **nous évitons tout changement dans `ReservationService` lorsque les restrictions d'accès sont modifiées.**
- **Nous ne `date_from` pas les données liées à l'entreprise (`date_from` , `date_to` , `reservations_count`) avec les concepts non liés au domaine (autorisations utilisateur) en service.**
- **Consumer (`StatsService`) est également exempt de logique liée aux autorisations**

CAVEATS

- L'interface proxy est toujours exactement identique à l'objet qu'elle cache, de sorte que l'utilisateur qui consomme le service encapsulé par un proxy n'était même pas au courant de la présence de proxy.

Lire Modèles de conception en ligne: https://riptutorial.com/fr/python/topic/8056/modeles-de-conception

Chapitre 118: Modèles en python

Examples

Programme de sortie de données simple utilisant un modèle

```
from string import Template

data = dict(item = "candy", price = 8, qty = 2)

# define the template
t = Template("Simon bought $qty $item for $price dollar")
print(t.substitute(data))
```

Sortie:

```
Simon bought 2 candy for 8 dollar
```

Les modèles prennent en charge les substitutions basées sur $ au lieu de la substitution basée sur le pourcentage. **Un substitut** (mappage, mots-clés) effectue une substitution de modèle, renvoyant une nouvelle chaîne.

Le mappage est un objet de type dictionnaire avec des clés correspondant aux espaces réservés du modèle. Dans cet exemple, le prix et la quantité sont des espaces réservés. Les arguments de mot clé peuvent également être utilisés comme des espaces réservés. Les espaces réservés des mots-clés ont priorité si les deux sont présents.

Changer le délimiteur

Vous pouvez remplacer le délimiteur "$" par un autre. L'exemple suivant:

```
from string import Template

class MyOtherTemplate(Template):
    delimiter = "#"


data = dict(id = 1, name = "Ricardo")
t = MyOtherTemplate("My name is #name and I have the id: #id")
print(t.substitute(data))
```

Vous pouvez lire de docs ici

Lire Modèles en python en ligne: https://riptutorial.com/fr/python/topic/6029/modeles-en-python

Chapitre 119: Module aléatoire

Syntaxe

- random.seed (a = None, version = 2) (la version est uniquement disponible pour python 3.x)
- random.getstate ()
- random.setstate (state)
- random.randint (a, b)
- random.randrange (stop)
- random.randrange (démarrer, arrêter, étape = 1)
- random.choice (seq)
- random.shuffle (x, random = random.random)
- random.sample (population, k)

Examples

Aléatoire et séquences: aléatoire, choix et échantillon

```
import random
```

mélanger ()

Vous pouvez utiliser `random.shuffle()` pour mélanger / randomiser les éléments dans une séquence **mutable et indexable** . Par exemple une `list` :

```
laughs = ["Hi", "Ho", "He"]

random.shuffle(laughs)     # Shuffles in-place! Don't do: laughs = random.shuffle(laughs)

print(laughs)
# Out: ["He", "Hi", "Ho"]  # Output may vary!
```

choix()

Prend un élément aléatoire d'une **séquence** arbitraire:

```
print(random.choice(laughs))
# Out: He                  # Output may vary!
```

échantillon()

Comme le `choice` il prend des éléments aléatoires d'une **séquence** arbitraire **,** mais vous pouvez spécifier combien:

```
#                    |--sequence--|--number--|
print(random.sample(    laughs    ,     1    ))  # Take one element
# Out: ['Ho']                     # Output may vary!
```

il ne faudra pas deux fois le même élément:

```
print(random.sample(laughs, 3))  # Take 3 random element from the sequence.
# Out: ['Ho', 'He', 'Hi']        # Output may vary!

print(random.sample(laughs, 4))  # Take 4 random element from the 3-item sequence.
```

ValueError: Échantillon plus grand que la population

Création d'entiers et de flottants aléatoires: randint, randrange, random et uniform

```
import random
```

randint ()

Retourne un entier aléatoire entre `x` et `y` (inclus):

```
random.randint(x, y)
```

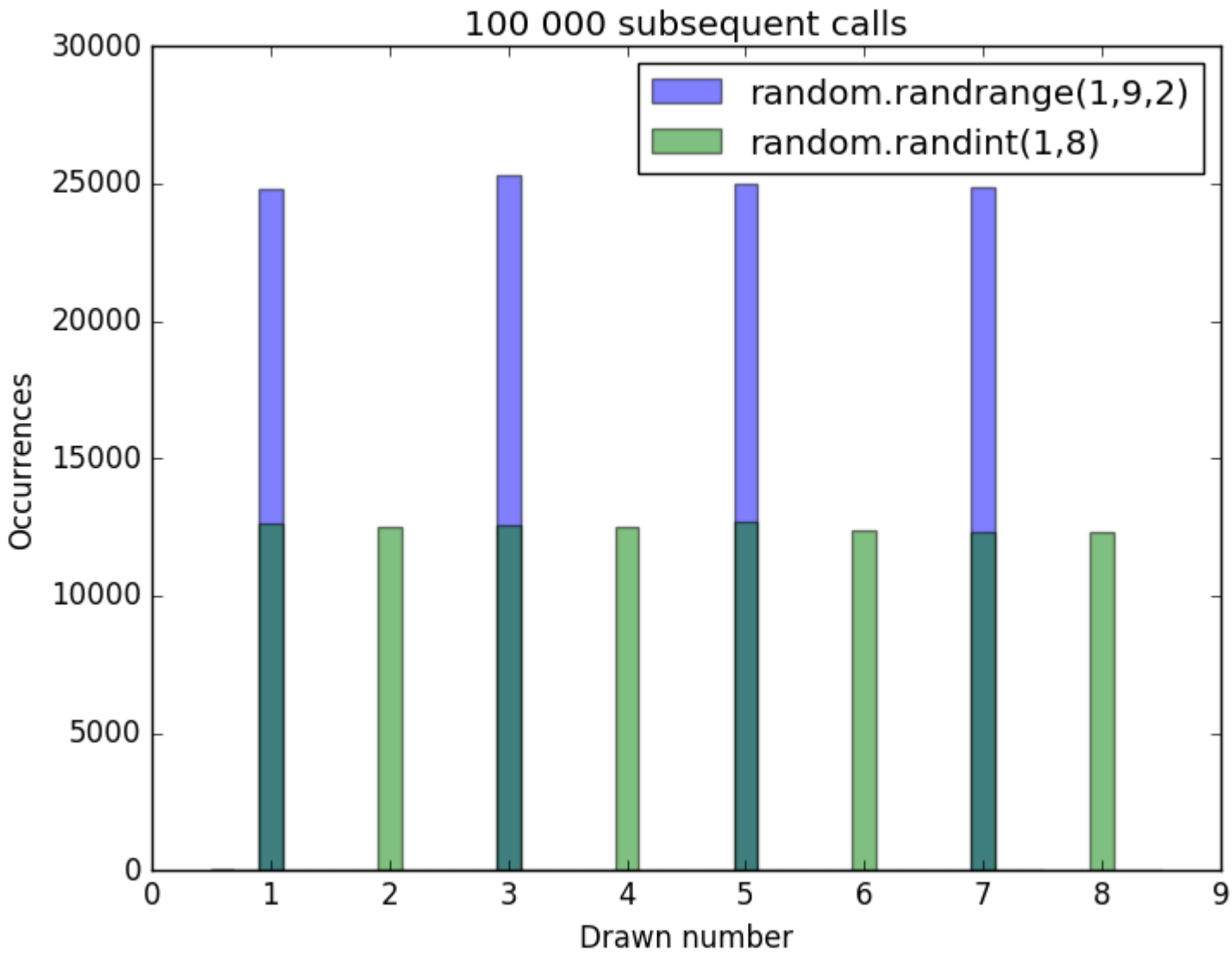
Par exemple obtenir un nombre aléatoire entre `1` et `8` :

```
random.randint(1, 8) # Out: 8
```

randrange ()

`random.randrange` a la même syntaxe que `range` et contrairement à `random.randint` , la dernière valeur n'est **pas** inclusive:

```
random.randrange(100)       # Random integer between 0 and 99
random.randrange(20, 50)    # Random integer between 20 and 49
random.rangrange(10, 20, 3) # Random integer between 10 and 19 with step 3 (10, 13, 16 and 19)
```



au hasard

Renvoie un nombre aléatoire en virgule flottante compris entre 0 et 1:

```
random.random() # Out: 0.66486093215306317
```

uniforme

Renvoie un nombre aléatoire en virgule flottante compris entre `x` et `y` (inclus):

```
random.uniform(1, 8) # Out: 3.726062641730108
```

Nombres aléatoires reproductibles: Semences et état

Définir une graine spécifique créera une série de nombres aléatoires fixes:

```
random.seed(5)                  # Create a fixed state
print(random.randrange(0, 10))  # Get a random integer between 0 and 9
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Réinitialiser la graine créera à nouveau la même séquence "aléatoire":

```
random.seed(5)                  # Reset the random module to the same fixed state.
print(random.randrange(0, 10))
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Comme la graine est fixée, ces résultats sont toujours `9` et `4` . Si des nombres spécifiques ne sont pas requis uniquement pour que les valeurs soient les mêmes, vous pouvez également utiliser `getstate` et `setstate` pour récupérer un état précédent:

```
save_state = random.getstate()  # Get the current state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8

random.setstate(save_state)     # Reset to saved state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8
```

Pour pseudo-randomiser la séquence à nouveau, vous `seed` avec `None` :

```
random.seed(None)
```

Ou appelez la méthode `seed` sans arguments:

```
random.seed()
```

Créer des nombres aléatoires sécurisés par cryptographie

Par défaut, le module aléatoire Python utilise le Mersenne Twister `PRNG` pour générer des nombres aléatoires qui, bien qu'adaptés à des domaines tels que les simulations, ne répondent pas aux exigences de sécurité dans des environnements plus exigeants.

Pour créer un nombre pseudo-aléatoire sécurisé sur le plan cryptographique, on peut utiliser `SystemRandom` qui, en utilisant `os.urandom` , peut agir en tant que générateur de nombres pseudo-aléatoires sécurisés par cryptographie, CPRNG .

La manière la plus simple de l'utiliser consiste simplement à initialiser la classe `SystemRandom` . Les méthodes fournies sont similaires à celles exportées par le module aléatoire.

```
from random import SystemRandom
secure_rand_gen = SystemRandom()
```

Pour créer une séquence aléatoire de 10 `int` s dans l'intervalle `[0, 20]` , on peut simplement appeler `randrange()` :

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

Pour créer un entier aléatoire dans une plage donnée, on peut utiliser `randint` :

```
print(secure_rand_gen.randint(0, 20))
# 5
```

et, en conséquence, pour toutes les autres méthodes. L'interface est exactement la même, le seul changement est le générateur de nombres sous-jacent.

Vous pouvez également utiliser `os.urandom` directement pour obtenir des octets aléatoires sécurisés par cryptographie.

Création d'un mot de passe utilisateur aléatoire

Afin de créer un mot de passe utilisateur aléatoire, nous pouvons utiliser les symboles fournis dans le module de `string` . Spécifiquement la `punctuation` pour les `punctuation` de ponctuation, les lettres `ascii_letters` pour les lettres et les `digits` pour les chiffres:

```
from string import punctuation, ascii_letters, digits
```

Nous pouvons alors combiner tous ces symboles dans un nom nommé `symbols` :

```
symbols = ascii_letters + digits + punctuation
```

Supprimez l'un de ces éléments pour créer un pool de symboles contenant moins d'éléments.

Après cela, nous pouvons utiliser `random.SystemRandom` pour générer un mot de passe. Pour un mot de passe de longueur 10:

```
secure_random = random.SystemRandom()
password = "".join(secure_random.choice(symbols) for i in range(10))
print(password)  # '^@g;J?]M6e'
```

Notez que les autres routines immédiatement disponibles par le module `random` - telles que `random.choice` , `random.randint` , etc. - ne *conviennent pas* à des fins de cryptographie.

Derrière les rideaux, ces routines utilisent le Mersenne Twister PRNG , qui ne répond pas aux exigences d'un CSPRNG . Ainsi, en particulier, vous ne devez en utiliser aucun pour générer des mots de passe que vous prévoyez d'utiliser. Utilisez toujours une instance de `SystemRandom` comme indiqué ci-dessus.

Python 3.x 3.6

À partir de Python 3.6, le module `secrets` est disponible, ce qui expose des fonctionnalités sécurisées sur le plan cryptographique.

En citant la documentation officielle , pour générer *"un mot de passe alphanumérique de dix caractères avec au moins un caractère minuscule, au moins une majuscule et au moins trois chiffres",* vous pouvez:

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Décision binaire aléatoire

```
import random

probability = 0.3

if random.random() < probability:
    print("Decision with probability 0.3")
else:
    print("Decision with probability 0.7")
```

Lire Module aléatoire en ligne: https://riptutorial.com/fr/python/topic/239/module-aleatoire

Chapitre 120: Module Asyncio

Examples

Coroutine et syntaxe de délégation

Avant la sortie de Python 3.5+, le module `asyncio` utilisait des générateurs pour imiter les appels asynchrones et avait donc une syntaxe différente de la version actuelle de Python 3.5.

Python 3.x 3.5

Python 3.5 a introduit l' `async` et `await` mots-clés. Notez l'absence de parenthèses autour de l'appel `await func()` .

```
import asyncio

async def main():
    print(await func())

async def func():
    # Do time intensive stuff...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x 3.3 3.5

Avant Python 3.5, le décorateur `@asyncio.coroutine` était utilisé pour définir une coroutine. Le rendement de l'expression a été utilisé pour la délégation du générateur. Notez les parenthèses autour du `yield from func()` .

```
import asyncio

@asyncio.coroutine
def main():
    print((yield from func()))

@asyncio.coroutine
def func():
    # Do time intensive stuff..
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x 3.5

Voici un exemple qui montre comment deux fonctions peuvent être exécutées de manière asynchrone:

```
import asyncio

async def cor1():
    print("cor1 start")
    for i in range(10):
        await asyncio.sleep(1.5)
        print("cor1", i)

async def cor2():
    print("cor2 start")
    for i in range(15):
        await asyncio.sleep(1)
        print("cor2", i)

loop = asyncio.get_event_loop()
cors = asyncio.wait([cor1(), cor2()])
loop.run_until_complete(cors)
```

Exécuteurs Asynchrones

Remarque: Utilise la syntaxe asynchrone / d'attente Python 3.5+

`asyncio` prend en charge l'utilisation des objets `Executor` trouvés dans `concurrent.futures` pour planifier des tâches de manière asynchrone. Les boucles d'événement ont la fonction `run_in_executor()` qui prend un objet `Executor` , un `Callable` et les paramètres de Callable.

Planification d'une tâche pour un `Executor`

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b

async def main(loop):
    executor = ThreadPoolExecutor()
    result = await loop.run_in_executor(executor, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

Chaque boucle d'événement dispose également d'un emplacement d' `Executor` "par défaut" pouvant être attribué à un `Executor` . Pour assigner un `Executor` et planifier des tâches depuis la boucle, vous utilisez la méthode `set_default_executor()` .

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b
```

```
async def main(loop):
    # NOTE: Using `None` as the first parameter designates the `default` Executor.
    result = await loop.run_in_executor(None, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.set_default_executor(ThreadPoolExecutor())
    loop.run_until_complete(main(loop))
```

Il existe deux principaux types d' `Executor` dans `concurrent.futures` , `ThreadPoolExecutor` et `ProcessPoolExecutor` . `ThreadPoolExecutor` contient un pool de threads pouvant être défini manuellement sur un nombre spécifique de threads via le constructeur ou par défaut sur le nombre de cœurs de la machine. `ThreadPoolExecutor` utilise le pool de threads pour exécuter les tâches qui lui sont affectées. généralement meilleure pour les opérations liées au processeur que pour les opérations liées aux E / S. Cela contraste avec `ProcessPoolExecutor` qui génère un nouveau processus pour chaque tâche qui lui est affectée. `ProcessPoolExecutor` ne peut que prendre en charge les tâches et les paramètres pouvant être capturés. Les tâches non-picklables les plus courantes sont les méthodes des objets. Si vous devez planifier la méthode d'un objet en tant que tâche dans un `Executor` vous devez utiliser un `ThreadPoolExecutor` .

Utiliser UVLoop

`uvloop` est une implémentation de `asyncio.AbstractEventLoop` basée sur libuv (Utilisé par nodejs). Il est compatible avec 99% des fonctionnalités `asyncio` et est beaucoup plus rapide que le traditionnel `asyncio.EventLoop` . `uvloop` n'est actuellement pas disponible sous Windows, installez-le avec `pip install uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Do your stuff here ...
```

On peut également modifier la fabrique de boucles d'événements en définissant `EventLoopPolicy` sur celle d' `uvloop` .

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

Primitive de synchronisation: événement

Concept

Utilisez un `Event` pour **synchroniser la planification de plusieurs coroutines** .

En d'autres termes, un événement est comme un coup de feu lors d'une course à pied: il permet aux coureurs de quitter les starting-blocks.

Exemple

```
import asyncio

# event trigger function
def trigger(event):
    print('EVENT SET')
    event.set()  # wake up coroutines waiting

# event consumers
async def consumer_a(event):
    consumer_name = 'Consumer A'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

async def consumer_b(event):
    consumer_name = 'Consumer B'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

# event
event = asyncio.Event()

# wrap coroutines in one future
main_future = asyncio.wait([consumer_a(event),
                            consumer_b(event)])

# event loop
event_loop = asyncio.get_event_loop()
event_loop.call_later(0.1, functools.partial(trigger, event))  # trigger event in 0.1 sec

# complete main_future
done, pending = event_loop.run_until_complete(main_future)
```

Sortie:

Consommateur B en attente
Un consommateur en attente
SET D'ÉVÉNEMENTS
Consommateur B déclenché
Consommateur A déclenché

Un websocket simple

Ici, nous faisons un websocket echo simple en utilisant `asyncio` . Nous définissons des coroutines pour se connecter à un serveur et envoyer / recevoir des messages. Les communications du

websocket sont exécutées dans une coroutine `main` , qui est exécutée par une boucle d'événement. Cet exemple est modifié depuis un article précédent .

```
import asyncio
import aiohttp

session = aiohttp.ClientSession()                              # handles the context manager
class EchoWebsocket:

    async def connect(self):
        self.websocket = await session.ws_connect("wss://echo.websocket.org")

    async def send(self, message):
        self.websocket.send_str(message)

    async def receive(self):
        result = (await self.websocket.receive())
        return result.data

async def main():
    echo = EchoWebsocket()
    await echo.connect()
    await echo.send("Hello World!")
    print(await echo.receive())                                # "Hello World!"


if __name__ == '__main__':
    # The main loop
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Idée commune à propos de l'asyncio

L'idée fausse *la* plus commune à propos de `asnycio` est `asnycio` qu'il vous permet d'exécuter n'importe quelle tâche en parallèle, en contournant le GIL (Global Interpreter Lock) et donc en exécutant des jobs de blocage en parallèle (sur des threads séparés). ça ne marche **pas** !

`asyncio` (et les bibliothèques `asyncio` pour collaborer avec `asyncio`) `asyncio` sur des coroutines: des fonctions qui (en collaboration) renvoient le flux de contrôle à la fonction appelante. Notez `asyncio.sleep` dans les exemples ci-dessus. Voici un exemple de coroutine non bloquante qui attend en arrière-plan et redonne le contrôle à la fonction appelante (lorsqu'elle est appelée avec `await`). `time.sleep` est un exemple de fonction de blocage. le flux d'exécution du programme s'arrêtera juste là et ne reviendra qu'après que `time.sleep` soit terminé.

un exemple réel est la bibliothèque de `requests` qui consiste (pour le moment) en fonctions de blocage uniquement. il n'y a pas de concurrence si vous appelez l'une de ses fonctions dans `asyncio` . `aiohttp` d'autre part a été construit avec `asyncio` à l'esprit. ses coroutines seront concurrentes.

- Si vous avez des tâches liées au processeur qui `asyncio` longtemps, que vous souhaitez exécuter en parallèle `asyncio` n'est **pas** pour vous. pour cela, vous avez besoin de `threads` ou de `multiprocessing` .

- Si vous avez des travaux liés aux E / S en cours d'exécution, vous *pouvez les* exécuter

simultanément en utilisant `asyncio` .

Lire Module Asyncio en ligne: https://riptutorial.com/fr/python/topic/1319/module-asyncio

Chapitre 121: Module Collections

Introduction

Le package de `collections` intégré fournit plusieurs types de collection spécialisés et flexibles, à la fois très performants et offrant des alternatives aux types de collection généraux de `dict` , `list` , `tuple` et `set` . Le module définit également des classes de base abstraites décrivant différents types de fonctionnalités de collecte (telles que `MutableSet` et `ItemsView`).

Remarques

Il existe trois autres types disponibles dans le module de **collections** , à savoir:

1. UserDict
2. Liste d'utilisateur
3. UserString

Ils agissent chacun comme une enveloppe autour de l'objet lié, par exemple, *UserDict* agit comme un wrapper autour d'un objet *dict* . Dans chaque cas, la classe simule son type nommé. Le contenu de l'instance est conservé dans un objet de type régulier, accessible via l'attribut data de l'instance wrapper. Dans chacun de ces trois cas, la nécessité de ces types a été partiellement remplacée par la possibilité de sous-classer directement à partir du type de base; Cependant, la classe wrapper peut être plus facile à utiliser car le type sous-jacent est accessible en tant qu'attribut.

Examples

collections.Counter

Counter est une sous-classe dict qui vous permet de compter facilement des objets. Il a des méthodes utilitaires pour travailler avec les fréquences des objets que vous comptez.

```
import collections
counts = collections.Counter([1,2,3])
```

le code ci-dessus crée un objet, compte, qui a les fréquences de tous les éléments passés au constructeur. Cet exemple a la valeur `Counter({1: 1, 2: 1, 3: 1})`

Exemples de constructeurs

Compteur de lettres

```
>>> collections.Counter('Happy Birthday')
Counter({'a': 2, 'p': 2, 'y': 2, 'i': 1, 'r': 1, 'B': 1, ' ': 1, 'H': 1, 'd': 1, 'h': 1, 't':
1})
```

Compteur de mots

```
>>> collections.Counter('I am Sam Sam I am That Sam-I-am That Sam-I-am! I do not like that
Sam-I-am'.split())
Counter({'I': 3, 'Sam': 2, 'Sam-I-am': 2, 'That': 2, 'am': 2, 'do': 1, 'Sam-I-am!': 1, 'that':
1, 'not': 1, 'like': 1})
```

Recettes

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

Obtenir le nombre d'éléments individuels

```
>>> c['a']
4
```

Définir le nombre d'éléments individuels

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

Obtenir le nombre total d'éléments dans le compteur (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues())  # negative numbers are counted!
3
```

Obtenir des éléments (seuls ceux avec un compteur positif sont conservés)

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Supprimer les clés avec une valeur 0 ou négative

```
>>> c - collections.Counter()
Counter({'a': 4, 'b': 2})
```

Tout enlever

```
>>> c.clear()
>>> c
Counter()
```

Ajouter supprimer des éléments individuels

```
>>> c.update({'a': 3, 'b':3})
>>> c.update({'a': 2, 'c':2})  # adds to existing, sets if they don't exist
>>> c
Counter({'a': 5, 'b': 3, 'c': 2})
>>> c.subtract({'a': 3, 'b': 3, 'c': 3})  # subtracts (negative values are allowed)
```

```
>>> c
Counter({'a': 2, 'b': 0, 'c': -1})
```

collections.defaultdict

collections.defaultdict (default_factory) renvoie une sous-classe de `dict` qui a une valeur par défaut pour les clés manquantes. L'argument doit être une fonction qui renvoie la valeur par défaut lorsqu'elle est appelée sans arguments. Si rien n'est passé, la valeur par défaut est `None` .

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

renvoie une référence à un defaultdict qui créera un objet de chaîne avec sa méthode default_factory.

Une utilisation typique de `defaultdict` est d'utiliser l'un des types intégrés tels que `str` , `int` , `list` ou `dict` comme default_factory, car ceux-ci renvoient des types vides lorsqu'ils sont appelés sans arguments:

```
>>> str()
''
>>> int()
0
>>> list
[]
```

L'appel du defaultdict avec une clé qui n'existe pas ne produit pas d'erreur comme dans un dictionnaire normal.

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

Un autre exemple avec `int` :

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2  # No errors should occur
>>> fruit_counts
default_dict(int, {'apple': 2})
>>> fruit_counts['banana']  # No errors should occur
0
>>> fruit_counts  # A new key is created
default_dict(int, {'apple': 2, 'banana': 0})
```

Les méthodes de dictionnaire normales fonctionnent avec le dictionnaire par défaut

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

Utiliser `list` comme default_factory va créer une liste pour chaque nouvelle clé.

```
>>> s = [('NC', 'Raleigh'), ('VA', 'Richmond'), ('WA', 'Seattle'), ('NC', 'Asheville')]
>>> dd = collections.defaultdict(list)
>>> for k, v in s:
...     dd[k].append(v)
>>> dd
defaultdict(<class 'list'>,
    {'VA': ['Richmond'],
     'NC': ['Raleigh', 'Asheville'],
     'WA': ['Seattle']})
```

collections.OrderedDict

L'ordre des clés dans les dictionnaires Python est arbitraire: elles ne sont pas régies par l'ordre dans lequel vous les ajoutez.

Par exemple:

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
```

(L'ordre arbitraire implicite ci-dessus signifie que vous pouvez obtenir des résultats différents avec le code ci-dessus à celui montré ici.)

L'ordre dans lequel les clés apparaissent est l'ordre dans lequel elles seront itérées, par exemple en utilisant une boucle `for` .

La classe `collections.OrderedDict` fournit des objets de dictionnaire qui conservent l'ordre des clés. `OrderedDict` peuvent être créées comme indiqué ci-dessous avec une série d'éléments ordonnés (ici, une liste de paires de clés-valeurs):

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('foo', 5), ('bar', 6)])
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6)])
>>> d['baz'] = 7
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7)])
>>> d['foobar'] = 8
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Ou nous pouvons créer un `OrderedDict` vide, puis ajouter des éléments:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

L'itération via un `OrderedDict` permet l'accès à la clé dans l'ordre dans `OrderedDict` ils ont été ajoutés.

Que se passe-t-il si nous attribuons une nouvelle valeur à une clé existante?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

La clé conserve sa place d'origine dans le `OrderedDict` .

collections.namedtuple

Définir un nouveau type `Person` utilisant `namedtuple` comme ceci:

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

Le second argument est la liste des attributs que le tuple aura. Vous pouvez répertorier ces attributs également sous forme de chaîne séparée par des espaces ou des virgules:

```
Person = namedtuple('Person', 'age, height, name')
```

ou

```
Person = namedtuple('Person', 'age height name')
```

Une fois défini, un tuple nommé peut être instancié en appelant l'objet avec les paramètres nécessaires, par exemple:

```
dave = Person(30, 178, 'Dave')
```

Les arguments nommés peuvent également être utilisés:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Vous pouvez maintenant accéder aux attributs du namedtuple:

```
print(jack.age)  # 30
print(jack.name)  # 'Jack S.'
```

Le premier argument du constructeur namedtuple (dans notre exemple `'Person'`) est le nom de `typename` . Il est courant d'utiliser le même mot pour le constructeur et le nom de fichier, mais ils

peuvent être différents:

```
Human = namedtuple('Person',  'age, height, name')
dave = Human(30, 178, 'Dave')
print(dave)  # yields: Person(age=30, height=178, name='Dave')
```

collections.deque

Retourne un nouvel objet `deque` initialisé de gauche à droite (en utilisant append ()) avec les données de l'itérable. Si itable n'est pas spécifié, le nouveau `deque` est vide.

Les Deques sont une généralisation des piles et des files d'attente (le nom se prononce «deck» et est l'abréviation de «double-queue queue»). Deques supportent ajouter ses thread-sûr, efficace et craquements mémoire de chaque côté de la `deque` avec approximativement la même O (1) la performance dans les deux sens.

Bien que les objets de liste prennent en charge des opérations similaires, ils sont optimisés pour les opérations rapides de longueur fixe et entraînent des coûts de mouvement de mémoire O (n) pour les opérations pop (0) et insert (0, v). .

Nouveau dans la version 2.4.

Si `maxlen` n'est pas spécifié ou est `None` , Deques peut atteindre une longueur arbitraire. Sinon, le `deque` est limité à la longueur maximale spécifiée. Une fois qu'un `deque` longueur limitée est plein, lorsque de nouveaux éléments sont ajoutés, un nombre correspondant d'éléments est éliminé de l'extrémité opposée. Les déques de longueur limitée offrent des fonctionnalités similaires à celles du filtre de queue sous Unix. Ils sont également utiles pour le suivi des transactions et des autres pools de données pour lesquels seule l’activité la plus récente présente un intérêt.

Modifié dans la version 2.6: Ajout du paramètre maxlen.

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
```

```
'i'

>>> list(reversed(d))               # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                        # search the deque
True
>>> d.extend('jkl')                 # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                     # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

Source: https://docs.python.org/2/library/collections.html

collections.ChainMap

`ChainMap` est nouveau dans la **version 3.3**

Renvoie un nouvel objet `ChainMap` plusieurs `maps` . Cet objet regroupe plusieurs dicts ou autres mappages pour créer une seule vue pouvant être mise à jour.

`ChainMap` s sont utiles pour gérer les contextes et les superpositions imbriqués. Un exemple dans le monde python se trouve dans l'implémentation de la classe `Context` dans le moteur de template de Django. Il est utile pour lier rapidement un certain nombre de mappages afin que le résultat puisse être traité comme une seule unité. Il est souvent beaucoup plus rapide que de créer un nouveau dictionnaire et d'exécuter plusieurs appels `update()` .

Chaque fois que l'on a une chaîne de valeurs de recherche, il peut y avoir un cas pour `ChainMap` . Un exemple consiste à avoir à la fois des valeurs spécifiées par l'utilisateur et un dictionnaire de valeurs par défaut. Un autre exemple est celui des cartes de paramètres `POST` et `GET` trouvées dans l'utilisation du Web, par exemple Django ou Flask. En utilisant `ChainMap` on renvoie une vue combinée de deux dictionnaires distincts.

La liste des paramètres des `maps` est triée de la première recherche à la dernière recherche. Les recherches recherchent les mappages sous-jacents successivement jusqu'à ce qu'une clé soit trouvée. En revanche, les écritures, les mises à jour et les suppressions ne fonctionnent que sur le premier mappage.

```
import collections

# define two dictionaries with at least some keys overlapping.
dict1 = {'apple': 1, 'banana': 2}
dict2 = {'coconut': 1, 'date': 1, 'apple': 3}

# create two ChainMaps with different ordering of those dicts.
combined_dict = collections.ChainMap(dict1, dict2)
reverse_ordered_dict = collections.ChainMap(dict2, dict1)
```

Notez l'impact de l'ordre sur lequel la valeur est trouvée en premier dans la recherche ultérieure

```
for k, v in combined_dict.items():
    print(k, v)

date 1
apple 1
banana 2
coconut 1

for k, v in reverse_ordered_dict.items():
    print(k, v)

date 1
apple 3
banana 2
coconut 1
```

Lire Module Collections en ligne: https://riptutorial.com/fr/python/topic/498/module-collections

Chapitre 122: Module de file d'attente

Introduction

Le module de file d'attente implémente des files d'attente multi-producteurs et multi-consommateurs. Il est particulièrement utile dans la programmation par thread lorsque des informations doivent être échangées en toute sécurité entre plusieurs threads. Il existe trois types de files d'attente par module de file d'attente, à savoir: 1. File d'attente 2. LifoQueue 3. Exception PriorityQueue pouvant être fournie: 1. Complet (dépassement de la file d'attente) 2. Vide (sous-dépassement de la file d'attente)

Examples

Exemple simple

```
from Queue import Queue

question_queue = Queue()

for x in range(1,10):
    temp_dict = ('key', x)
    question_queue.put(temp_dict)

while(not question_queue.empty()):
    item = question_queue.get()
    print(str(item))
```

Sortie:

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

Lire Module de file d'attente en ligne: https://riptutorial.com/fr/python/topic/8339/module-de-file-d-attente

Chapitre 123: Module Deque

Syntaxe

- dq = deque () # Crée un deque vide
- dq = deque (iterable) # Crée un deque avec des éléments
- dq.append (object) # Ajoute un objet à droite de la deque
- dq.appendleft (object) # Ajoute un objet à gauche de la deque
- dq.pop () -> objet # Supprime et retourne l'objet le plus à droite
- dq.popleft () -> object # Supprime et renvoie l'objet le plus à gauche
- dq.extend (itérable) # Ajoute quelques éléments à droite du deque
- dq.extendleft (iterable) # Ajoute des éléments à la gauche du deque

Paramètres

Paramètre	Détails
`iterable`	Crée le deque avec les éléments initiaux copiés à partir d'une autre itération.
`maxlen`	Limite la taille du déque, en repoussant les anciens éléments à mesure que de nouveaux éléments sont ajoutés.

Remarques

Cette classe est utile lorsque vous avez besoin d'un objet similaire à une liste qui permet des opérations rapides d'ajout et d'ajout d'un côté ou de l'autre (le nom `deque` signifie « *file d'attente double* »).

Les méthodes fournies sont en effet très similaires, sauf que certaines, comme `pop` , `append` ou `extend` peuvent être suffixées par `left` . La structure de données `deque` doit être préférée à une liste si l'on doit fréquemment insérer et supprimer des éléments aux deux extrémités car cela permet de le faire à temps constant O (1).

Examples

Deque de base en utilisant

Les principales méthodes utiles avec cette classe sont `popleft` et `appendleft`

```
from collections import deque

d = deque([1, 2, 3])
p = d.popleft()         # p = 1, d = deque([2, 3])
d.appendleft(5)         # d = deque([5, 2, 3])
```

limiter la taille de deque

Utilisez le paramètre `maxlen` lors de la création d'un deque pour limiter la taille du deque:

```
from collections import deque
d = deque(maxlen=3)  # only holds 3 items
d.append(1)  # deque([1])
d.append(2)  # deque([1, 2])
d.append(3)  # deque([1, 2, 3])
d.append(4)  # deque([2, 3, 4]) (1 is removed because its maxlen is 3)
```

Méthodes disponibles dans deque

Créer un deque vide:

```
dl = deque()  # deque([]) creating empty deque
```

Créer deque avec quelques éléments:

```
dl = deque([1, 2, 3, 4])  # deque([1, 2, 3, 4])
```

Ajouter un élément à deque:

```
dl.append(5)  # deque([1, 2, 3, 4, 5])
```

Ajout de l'élément côté gauche de deque:

```
dl.appendleft(0)  # deque([0, 1, 2, 3, 4, 5])
```

Ajouter une liste d'éléments à deque:

```
dl.extend([6, 7])  # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

Ajout de la liste des éléments du côté gauche:

```
dl.extendleft([-2, -1])  # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

L' `.pop()` élément `.pop()` supprime naturellement un élément du côté droit:

```
dl.pop()  # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

Utiliser l'élément `.popleft()` pour supprimer un élément du côté gauche:

```
dl.popleft()  # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

Supprimer l'élément par sa valeur:

```
dl.remove(1)  # deque([-2, 0, 2, 3, 4, 5, 6])
```

Inverser l'ordre des éléments dans deque:

```
dl.reverse()  # deque([6, 5, 4, 3, 2, 0, -2])
```

Largeur Première Recherche

Le Deque est la seule structure de données Python avec des opérations de file d'attente rapides. (Remarque `queue.Queue` n'est normalement pas approprié, car il est destiné à la communication entre les threads.) Un cas d'utilisation de base d'une file d'attente est la première recherche .

```
from collections import deque

def bfs(graph, root):
    distances = {}
    distances[root] = 0
    q = deque([root])
    while q:
        # The oldest seen (but not yet visited) node will be the left most one.
        current = q.popleft()
        for neighbor in graph[current]:
            if neighbor not in distances:
                distances[neighbor] = distances[current] + 1
                # When we see a new node, we add it to the right side of the queue.
                q.append(neighbor)
    return distances
```

Disons que nous avons un graphique simple dirigé:

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

Nous pouvons maintenant trouver les distances depuis une position de départ:

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

Lire Module Deque en ligne: https://riptutorial.com/fr/python/topic/1976/module-deque

Chapitre 124: Module Functools

Examples

partiel

La fonction `partial` crée une application de fonction partielle à partir d'une autre fonction. Il est utilisé pour *lier des* valeurs à certains des arguments de la fonction (ou des arguments de mots-clés) et produire un *appelable* sans les arguments déjà définis.

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('ca11ab1e')
3390155550
```

`partial()` , comme son nom l'indique, permet une évaluation partielle d'une fonction. Regardons l'exemple suivant:

```
In [2]: from functools import partial

In [3]: def f(a, b, c, x):
   ...:     return 1000*a + 100*b + 10*c + x
   ...:

In [4]: g = partial(f, 1, 1, 1)

In [5]: print g(2)
1112
```

Lorsque `g` est créé, `f` , qui prend quatre arguments (`a, b, c, x`), est aussi partiellement évalué pour les trois premiers arguments, `a, b, c,` L'évaluation de `f` est terminée lorsque `g` est appelé, `g(2)` , qui passe le quatrième argument à `f` .

Une manière de penser à `partial` est un registre à décalage; en poussant un argument à la fois dans une fonction. `partial` est utile pour les cas où les données entrent en flux et que nous ne pouvons pas passer plus d'un argument.

total_ordering

Lorsque nous voulons créer une classe ordonnable, nous devons normalement définir les méthodes `__eq()__` , `__lt__()` , `__le__()` , `__gt__()` et `__ge__()` .

Le décorateur `total_ordering` , appliqué à une classe, autorise la définition de `__eq__()` et une seule entre `__lt__()` , `__le__()` , `__gt__()` et `__ge__()` , tout en autorisant toutes les opérations de classement sur la classe.

```
@total_ordering
```

```
class Employee:

    ...

    def __eq__(self, other):
        return ((self.surname, self.name) == (other.surname, other.name))

    def __lt__(self, other):
        return ((self.surname, self.name) < (other.surname, other.name))
```

Le décorateur utilise une composition des méthodes fournies et des opérations algébriques pour dériver les autres méthodes de comparaison. Par exemple, si nous avons défini `__lt__()` et `__eq()__` et que nous voulons dériver `__gt__()` , nous pouvons simplement vérifier `not __lt__() and not __eq()__` .

Remarque : La fonction `total_ordering` est uniquement disponible depuis Python 2.7.

réduire

Dans Python 3.x, la fonction de `reduce` déjà expliquée ici a été supprimée des éléments intégrés et doit maintenant être importée à partir de `functools` .

```
from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))
```

lru_cache

Le décorateur `@lru_cache` peut être utilisé pour `@lru_cache` une fonction coûteuse et intensive en calcul avec un cache moins utilisé récemment . Cela permet de mémoriser les appels de fonctions, de sorte que les futurs appels avec les mêmes paramètres puissent revenir instantanément au lieu de devoir être recalculés.

```
@lru_cache(maxsize=None)  # Boundless cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)
```

Dans l'exemple ci-dessus, la valeur de `fibonacci(3)` n'est calculée qu'une seule fois, alors que si `fibonacci` n'avait pas de cache LRU, `fibonacci(3)` aurait été calculé plus de 230 fois. Par conséquent, `@lru_cache` est particulièrement `@lru_cache` aux fonctions récursives ou à la programmation dynamique, où une fonction coûteuse peut être appelée plusieurs fois avec les mêmes paramètres exacts.

`@lru_cache` a deux arguments

- `maxsize` : nombre d'appels à enregistrer. Lorsque le nombre d'appels uniques dépasse la `maxsize` , le cache LRU supprime les appels les moins récemment utilisés.

- `typed` (ajouté en 3.3): Indicateur permettant de déterminer si des arguments équivalents de différents types appartiennent à des enregistrements de cache différents (par exemple, si les arguments `3.0` et `3` sont différents)

Nous pouvons également voir les statistiques du cache:

```
>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

REMARQUE : Comme `@lru_cache` utilise des dictionnaires pour mettre en cache les résultats, tous les paramètres de la fonction doivent être gérables pour que le cache fonctionne.

`@lru_cache` Python officiels pour `@lru_cache` . `@lru_cache` été ajouté en 3.2.

cmp_to_key

Python a changé ses méthodes de tri pour accepter une fonction clé. Ces fonctions prennent une valeur et renvoient une clé utilisée pour trier les tableaux.

Les anciennes fonctions de comparaison utilisaient deux valeurs et renvoyaient -1, 0 ou +1 si le premier argument était petit, égal ou supérieur au second argument, respectivement. Ceci est incompatible avec la nouvelle fonction clé.

C'est là `functools.cmp_to_key` :

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

Exemple pris et adapté de la documentation Python Standard Library .

Lire Module Functools en ligne: https://riptutorial.com/fr/python/topic/2492/module-functools

Chapitre 125: Module Itertools

Syntaxe

- `import itertools`

Examples

Regroupement d'éléments à partir d'un objet pouvant être itéré à l'aide d'une fonction

Commencez avec une itération qui doit être regroupée

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

Générer le générateur groupé, regroupé par le deuxième élément de chaque tuple:

```
def testGroupBy(lst):
    groups = itertools.groupby(lst, key=lambda x: x[1])
    for key, group in groups:
        print(key, list(group))

testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
```

Seuls les groupes d'éléments consécutifs sont regroupés. Vous devrez peut-être trier par la même clé avant d'appeler groupby For Eg (le dernier élément est modifié)

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 5, 6)]
testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5)]
# 5 [('c', 5, 6)]
```

Le groupe renvoyé par groupby est un itérateur qui sera invalide avant la prochaine itération. Par exemple, les éléments suivants ne fonctionneront pas si vous souhaitez que les groupes soient triés par clé. Le groupe 5 est vide ci-dessous car lorsque le groupe 2 est récupéré, il invalide 5

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted(groups):
    print(key, list(group))

# 2 [('c', 2, 6)]
# 5 []
```

Pour effectuer correctement le tri, créez une liste à partir de l'itérateur avant de trier

```
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted((key, list(group)) for key, group in groups):
    print(key, list(group))

# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
# 5 [('a', 5, 6)]
```

Prendre une tranche de générateur

Itertools "islice" vous permet de découper un générateur:

```
results = fetch_paged_results()  # returns a generator
limit = 20  # Only want the first 20 results
for data in itertools.islice(results, limit):
    print(data)
```

Normalement, vous ne pouvez pas découper un générateur:

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[:3]:
    print(part)
```

Va donner

```
Traceback (most recent call last):
  File "gen.py", line 6, in <module>
    for part in gen()[:3]:
TypeError: 'generator' object is not subscriptable
```

Cependant, cela fonctionne:

```
import itertools

def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in itertools.islice(gen(), 3):
    print(part)
```

Notez que, comme une tranche normale, vous pouvez également utiliser les arguments `start` , `stop` et `step` :

```
itertools.islice(iterable, 1, 30, 3)
```

itertools.product

Cette fonction vous permet de parcourir le produit cartésien d'une liste d'itérables.

Par exemple,

```
for x, y in itertools.product(xrange(10), xrange(10)):
    print x, y
```

est équivalent à

```
for x in xrange(10):
    for y in xrange(10):
        print x, y
```

Comme toutes les fonctions python acceptant un nombre variable d'arguments, nous pouvons passer une liste à itertools.product pour le décompresser, avec l'opérateur *.

Ainsi,

```
its = [xrange(10)] * 2
for x,y in itertools.product(*its):
    print x, y
```

produit les mêmes résultats que les deux exemples précédents.

```
>>> from itertools import product
>>> a=[1,2,3,4]
>>> b=['a','b','c']
>>> product(a,b)
<itertools.product object at 0x0000000002712F78>
>>> for i in product(a,b):
...     print i
...
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
(4, 'a')
(4, 'b')
(4, 'c')
```

itertools.count

Introduction:

Cette fonction simple génère une série infinie de nombres. Par exemple...

```
for number in itertools.count():
    if number > 20:
        break
    print(number)
```

Notez que nous devons casser ou imprimer pour toujours!

Sortie:

```
0
1
2
3
4
5
6
7
8
9
10
```

Arguments:

`count()` prend deux arguments, `start` et `step` :

```
for number in itertools.count(start=10, step=4):
    print(number)
    if number > 20:
        break
```

Sortie:

```
10
14
18
22
```

itertools.takewhile

itertools.takewhile vous permet de prendre des éléments d'une séquence jusqu'à ce qu'une condition devienne la première `False` .

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.takewhile(is_even, lst))

print(result)
```

Cette sortie `[0, 2, 4, 12, 18]` .

Notez que le premier nombre qui viole le prédicat (c'est-à-dire la fonction renvoyant une valeur booléenne) `is_even` est `13` . Une fois que `takewhile` rencontre une valeur qui produit `False` pour le prédicat donné, elle éclate.

La **sortie produite** par `takewhile` est similaire à la sortie générée par le code ci-dessous.

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Note: La concaténation des résultats produits par `takewhile` et `dropwhile` produit l'itérable original.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

itertools.dropwhile

itertools.dropwhile vous permet de prendre des éléments d'une séquence après qu'une condition soit devenue la première `False` .

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.dropwhile(is_even, lst))

print(result)
```

Cette sortie `[13, 14, 22, 23, 44]` .

(*Cet exemple est identique à l'exemple de* `takewhile` *mais en utilisant* `dropwhile` *.*)

Notez que le premier nombre qui viole le prédicat (c'est-à-dire la fonction renvoyant une valeur booléenne) `is_even` est `13` . Tous les éléments avant cela sont éliminés.

La **sortie produite** par `dropwhile` est similaire à la sortie générée par le code ci-dessous.

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

La concaténation des résultats produits par `takewhile` et `dropwhile` produit l'itérable original.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

Zipper deux itérateurs jusqu'à ce qu'ils soient tous deux épuisés

Semblable à la fonction intégrée `zip()` , `itertools.zip_longest` continuera à itérer au-delà de la plus courte des deux itérables.

```
from itertools import zip_longest
a = [i for i in range(5)] # Length is 5
b = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # Length is 7
for i in zip_longest(a, b):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

Un argument facultatif `fillvalue` peut être passé (par défaut à `''`) comme ceci:

```
for i in zip_longest(a, b, fillvalue='Hogwash!'):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

Dans Python 2.6 et 2.7, cette fonction s'appelle `itertools.izip_longest` .

Méthode des combinaisons dans le module Itertools

`itertools.combinations` retournera un générateur de la séquence de combinaison *k* d'une liste.

En d'autres termes: il renverra un générateur de tuples de toutes les combinaisons possibles en k de la liste d'entrée.

Par exemple:

Si vous avez une liste:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 2))
print b
```

Sortie:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

La sortie ci - dessus est un générateur converti en une liste d'uplets de l' ensemble de la *paire* possible -wise combinaisons de la liste d'entrée d' `a`

Vous pouvez également trouver toutes les 3 combinaisons:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 3))
print b
```

Sortie:

```
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4),
```

```
 (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5),
 (2, 4, 5), (3, 4, 5)]
```

Enchaînement multiple d'itérateurs

Utilisez `itertools.chain` pour créer un seul générateur qui produira les valeurs de plusieurs générateurs en séquence.

```
from itertools import chain
a = (x for x in ['1', '2', '3', '4'])
b = (x for x in ['x', 'y', 'z'])
' '.join(chain(a, b))
```

Résulte en:

```
'1 2 3 4 x y z'
```

En tant que constructeur alternatif, vous pouvez utiliser classmethod `chain.from_iterable` qui prend comme paramètre unique une itération d'itérables. Pour obtenir le même résultat que ci-dessus:

```
' '.join(chain.from_iterable([a,b])
```

Alors que la `chain` peut prendre un nombre arbitraire d'arguments, `chain.from_iterable` est le seul moyen de chaîner un nombre *infini* d'itérables.

itertools.reat

Répétez quelque chose n fois:

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...     print(i)
over-and-over
over-and-over
over-and-over
```

Obtenir une somme cumulée de nombres dans une itération

Python 3.x 3.2

`accumulate` donne une somme (ou un produit) cumulatif de nombres.

```
>>> import itertools as it
>>> import operator

>>> list(it.accumulate([1,2,3,4,5]))
[1, 3, 6, 10, 15]

>>> list(it.accumulate([1,2,3,4,5], func=operator.mul))
[1, 2, 6, 24, 120]
```

Parcourir des éléments dans un itérateur

`cycle` est un itérateur infini.

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

Par conséquent, veillez à donner des limites lorsque vous utilisez ceci pour éviter une boucle infinie. Exemple:

```
>>> # Iterate over each element in cycle for a fixed range
>>> cycle_iterator = it.cycle('abc123')
>>> [next(cycle_iterator) for i in range(0, 10)]
['a', 'b', 'c', '1', '2', '3', 'a', 'b', 'c', '1']
```

itertools.permutations

`itertools.permutations` renvoie un générateur avec des permutations successives de longueur r d'éléments dans l'itérable.

```
a = [1,2,3]
list(itertools.permutations(a))
# [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]

list(itertools.permutations(a, 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

si la liste `a` comporte des éléments en double, les permutations résultants ont des éléments en double, vous pouvez utiliser `set` pour obtenir des permutations uniques:

```
a = [1,2,1]
list(itertools.permutations(a))
# [(1, 2, 1), (1, 1, 2), (2, 1, 1), (2, 1, 1), (1, 1, 2), (1, 2, 1)]

set(itertools.permutations(a))
# {(1, 1, 2), (1, 2, 1), (2, 1, 1)}
```

Lire Module Itertools en ligne: https://riptutorial.com/fr/python/topic/1564/module-itertools

Chapitre 126: Module JSON

Remarques

Pour une documentation complète, y compris les fonctionnalités spécifiques à la version, consultez la documentation officielle .

Les types

Les défauts

le module `json` gérera par défaut l'encodage et le décodage des types ci-dessous:

Types de désérialisation:

JSON	Python
objet	dict
tableau	liste
chaîne	str
nombre (int)	int
nombre (réel)	flotte
vrai faux	Vrai faux
nul	Aucun

Le module `json` comprend également `NaN` , `Infinity` et `-Infinity` comme valeurs flottantes correspondantes, ce qui est en dehors de la spécification JSON.

Types de sérialisation:

Python	JSON
dict	objet
liste, tuple	tableau
str	chaîne
int, float, (int / float) -enums dérivés	nombre

Python	JSON
Vrai	vrai
Faux	faux
Aucun	nul

Pour interdire le codage de `NaN` , `Infinity` et `-Infinity` vous devez encoder avec `allow_nan=False` . Cela `ValueError` alors une valeur `ValueError` si vous tentez de coder ces valeurs.

Sérialisation personnalisée

Il existe différents hooks qui vous permettent de gérer des données qui doivent être représentées différemment. L'utilisation de `functools.partial` vous permet d'appliquer partiellement les paramètres pertinents à ces fonctions pour plus de commodité.

Sérialisation:

Vous pouvez fournir une fonction qui fonctionne sur les objets avant qu'ils ne soient sérialisés comme suit:

```
# my_json module

import json
from functools import partial

def serialise_object(obj):
    # Do something to produce json-serialisable data
    return dict_obj

dump = partial(json.dump, default=serialise_object)
dumps = partial(json.dumps, default=serialise_object)
```

Désérialisation:

Il existe différents hooks gérés par les fonctions json, tels que object_hook et parse_float. Pour une liste exhaustive de votre version de python, voir ici .

```
# my_json module

import json
from functools import partial

def deserialise_object(dict_obj):
    # Do something custom
    return obj

def deserialise_float(str_obj):
    # Do something custom
    return obj
```

```
load = partial(json.load, object_hook=deserialise_object, parse_float=deserialise_float)
loads = partial(json.loads, object_hook=deserialise_object, parse_float=deserialise_float)
```

Sérialisation (dé) personnalisée supplémentaire:

Le module `json` permet également l'extension / substitution de `json.JSONEncoder` et `json.JSONDecoder` pour gérer des types divers. Les hooks documentés ci-dessus peuvent être ajoutés par défaut en créant une méthode nommée de manière équivalente. Pour les utiliser, il suffit de passer la classe en tant que paramètre `cls` à la fonction concernée. L'utilisation de `functools.partial` vous permet d'appliquer partiellement le paramètre cls à ces fonctions, par exemple

```
# my_json module

import json
from functools import partial

class MyEncoder(json.JSONEncoder):
    # Do something custom

class MyDecoder(json.JSONDecoder):
    # Do something custom

dump = partial(json.dump, cls=MyEncoder)
dumps = partial(json.dumps, cls=MyEncoder)
load = partial(json.load, cls=MyDecoder)
loads = partial(json.loads, cls=MyDecoder)
```

Examples

Création de JSON à partir de Python dict

```
import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)
```

L'extrait ci-dessus renvoie les éléments suivants:

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

Créer un dict Python depuis JSON

```
import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)
```

L'extrait ci-dessus renvoie les éléments suivants:

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

Stocker des données dans un fichier

L'extrait de code suivant code les données stockées dans `d` en JSON et les stocke dans un fichier (remplace le `filename` par le nom réel du fichier).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

Récupération des données d'un fichier

L'extrait de code suivant ouvre un fichier codé JSON (remplace le `filename` de `filename` par le nom réel du fichier) et renvoie l'objet stocké dans le fichier.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

`load` vs` charges`, `dump` vs` dumps`

Le module `json` contient des fonctions à la fois pour lire et écrire depuis et vers des chaînes Unicode, ainsi que pour lire et écrire depuis et vers des fichiers. Celles-ci sont différenciées par un `s` dans le nom de la fonction. Dans ces exemples, nous utilisons un objet StringIO, mais les mêmes fonctions s'appliqueraient à tout objet de type fichier.

Ici, nous utilisons les fonctions basées sur les chaînes:

```
import json

data = {u"foo": u"bar", u"baz": []}
json_string = json.dumps(data)
# u'{"foo": "bar", "baz": []}'
json.loads(json_string)
# {u"foo": u"bar", u"baz": []}
```

Et ici, nous utilisons les fonctions basées sur les fichiers:

```
import json

from io import StringIO
```

```
json_file = StringIO()
data = {u"foo": u"bar", u"baz": []}
json.dump(data, json_file)
json_file.seek(0)  # Seek back to the start of the file before reading
json_file_content = json_file.read()
# u'{"foo": "bar", "baz": []}'
json_file.seek(0)  # Seek back to the start of the file before reading
json.load(json_file)
# {u"foo": u"bar", u"baz": []}
```

Comme vous pouvez le constater, la principale différence est que lors du vidage de données json, vous devez transmettre le descripteur de fichier à la fonction, par opposition à la capture de la valeur de retour. Il convient également de noter que vous devez chercher au début du fichier avant de lire ou d'écrire afin d'éviter toute corruption des données. Lorsque vous ouvrez un fichier, le curseur est placé à la position `0` , de sorte que ce qui suit fonctionnerait également:

```
import json

json_file_path = './data.json'
data = {u"foo": u"bar", u"baz": []}

with open(json_file_path, 'w') as json_file:
    json.dump(data, json_file)

with open(json_file_path) as json_file:
    json_file_content = json_file.read()
    # u'{"foo": "bar", "baz": []}'

with open(json_file_path) as json_file:
    json.load(json_file)
    # {u"foo": u"bar", u"baz": []}
```

Avoir les deux manières de gérer les données json vous permet de travailler de manière idiomatique et efficace avec des formats `pyspark` sur json, tels que json-par-line de `pyspark` :

```
# loading from a file
data = [json.loads(line) for line in open(file_path).splitlines()]

# dumping to a file
with open(file_path, 'w') as json_file:
    for item in data:
        json.dump(item, json_file)
        json_file.write('\n')
```

Appeler `json.tool` depuis la ligne de commande pour imprimer joliment la sortie JSON

Étant donné un fichier JSON "foo.json" comme:

```
{"foo": {"bar": {"baz": 1}}}
```

on peut appeler le module directement depuis la ligne de commande (en passant le nom du fichier en argument) pour le faire imprimer:

```
$ python -m json.tool foo.json
{
    "foo": {
        "bar": {
            "baz": 1
        }
    }
}
```

Le module prendra également en compte STDOUT, donc (dans Bash), nous pourrions tout aussi bien:

```
$ cat foo.json | python -m json.tool
```

Formatage de la sortie JSON

Disons que nous avons les données suivantes:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color":
"black"}]}
```

Json ne fait rien de spécial ici:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Définition de l'indentation pour obtenir une sortie plus jolie

Si nous voulons une jolie impression, nous pouvons définir une taille de `indent` :

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```

Trier les clés par ordre alphabétique pour

obtenir une sortie cohérente

Par défaut, l'ordre des clés dans la sortie est indéfini. Nous pouvons les obtenir par ordre alphabétique pour nous assurer que nous obtenons toujours le même résultat:

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

Se débarrasser des espaces pour obtenir une sortie compacte

Nous pourrions vouloir supprimer les espaces inutiles, ce qui est fait en définissant des chaînes de séparation différentes des valeurs par défaut `', '` et `': '` :

```
>>>print(json.dumps(data, separators=(',', ':')))
{"cats":[{"name":"Tubbs","color":"white"},{"name":"Pepper","color":"black"}]}
```

JSON codant des objets personnalisés

Si nous essayons juste ce qui suit:

```
import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))
```

nous obtenons une erreur en disant `TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializable` .

Pour pouvoir sérialiser correctement l'objet datetime, nous devons écrire un code personnalisé pour le convertir:

```
class DatetimeJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        try:
            return obj.isoformat()
        except AttributeError:
            # obj has no isoformat method; let the builtin JSON encoder handle it
            return super(DatetimeJSONEncoder, self).default(obj)
```

puis utilisez cette classe d'encodeur au lieu de `json.dumps` :

```
encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# prints {"datetime": "2016-09-26T04:44:00"}
```

Lire Module JSON en ligne: https://riptutorial.com/fr/python/topic/272/module-json

Chapitre 127: Module Math

Examples

Arrondi: rond, sol, plafond, tronc

Outre la fonction `round` intégrée, le module `math` fournit les fonctions `floor` , `ceil` et `trunc` .

```
x = 1.55
y = -1.55

# round to the nearest integer
round(x)       #  2
round(y)       # -2

# the second argument gives how many decimal places to round to (defaults to 0)
round(x, 1)    #  1.6
round(y, 1)    # -1.6

# math is a module so import it first, then use it.
import math

# get the largest integer less than x
math.floor(x)  #  1
math.floor(y)  # -2

# get the smallest integer greater than x
math.ceil(x)   #  2
math.ceil(y)   # -1

# drop fractional part of x
math.trunc(x)  #  1, equivalent to math.floor for positive numbers
math.trunc(y)  # -1, equivalent to math.ceil for negative numbers
```

Python 2.x 2.7

`floor` , `ceil` , `trunc` et `round` retournent toujours un `float` .

```
round(1.3)  # 1.0
```

`round` casse toujours les liens à partir de zéro.

```
round(0.5)  # 1.0
round(1.5)  # 2.0
```

Python 3.x 3.0

`floor` , `ceil` et `trunc` renvoient toujours une valeur `Integral` , alors que `round` renvoie une valeur `Integral` si elle est appelée avec un argument.

```
round(1.3)      # 1
round(1.33, 1)  # 1.3
```

`round` casse les liens vers le nombre pair le plus proche. Cela corrige le biais vers de plus grands nombres lors de l'exécution d'un grand nombre de calculs.

```
round(0.5)  # 0
round(1.5)  # 2
```

Attention!

Comme pour toute représentation en virgule flottante, certaines fractions *ne peuvent pas être représentées exactement* . Cela peut entraîner un comportement d'arrondi inattendu.

```
round(2.675, 2)  # 2.67, not 2.68!
```

Avertissement concernant la division floor, trunc et entier des nombres négatifs

Python (et C ++ et Java) arrondit à zéro pour les nombres négatifs. Considérer:

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

Logarithmes

`math.log(x)` donne le logarithme naturel (base `e`) de `x` .

```
math.log(math.e)  # 1.0
math.log(1)       # 0.0
math.log(100)     # 4.605170185988092
```

`math.log` peut perdre de la précision avec des nombres proches de 1, en raison des limitations des nombres à virgule flottante. Pour calculer avec précision les journaux proches de 1, utilisez `math.log1p` , qui évalue le logarithme naturel de 1 plus l'argument:

```
math.log(1 + 1e-20)  # 0.0
math.log1p(1e-20)    # 1e-20
```

`math.log10` peut être utilisé pour la base de journaux 10:

```
math.log10(10)  # 1.0
```

Python 2.x 2.3.0

Lorsqu'il est utilisé avec deux arguments, `math.log(x, base)` donne le logarithme de `x` dans la `base` donnée (c'est-à-dire `log(x) / log(base)` .

```
math.log(100, 10) # 2.0
```

```
math.log(27, 3)   # 3.0
math.log(1, 10)   # 0.0
```

Signes de copie

Dans Python 2.6 et `math.copysign(x, y)` ultérieures, `math.copysign(x, y)` renvoie `x` avec le signe de `y` . La valeur renvoyée est toujours un `float` .

Python 2.x 2.6

```
math.copysign(-2, 3)    # 2.0
math.copysign(3, -3)    # -3.0
math.copysign(4, 14.2)  # 4.0
math.copysign(1, -0.0)  # -1.0, on a platform which supports signed zero
```

Trigonométrie

Calcul de la longueur de l'hypoténuse

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)
# Out: 4.47213595499958
```

Conversion de degrés en radians

Toutes `math` fonctions `math` attendent des **radians** , vous devez donc convertir les degrés en radians:

```
math.radians(45)              # Convert 45 degrees to radians
# Out: 0.7853981633974483
```

Tous les résultats des fonctions trigonométriques inverses renvoient le résultat en radians, vous devrez donc le convertir en degrés:

```
math.degrees(math.asin(1))    # Convert the result of asin to degrees
# Out: 90.0
```

Fonctions sinus, cosinus, tangente et inverse

```
# Sine and arc sine
math.sin(math.pi / 2)
# Out: 1.0
math.sin(math.radians(90))   # Sine of 90 degrees
# Out: 1.0

math.asin(1)
# Out: 1.5707963267948966    # "= pi / 2"
math.asin(1) / math.pi
# Out: 0.5

# Cosine and arc cosine:
```

```
math.cos(math.pi / 2)
# Out: 6.123233995736766e-17
# Almost zero but not exactly because "pi" is a float with limited precision!

math.acos(1)
# Out: 0.0

# Tangent and arc tangent:
math.tan(math.pi/2)
# Out: 1.633123935319537e+16
# Very large but not exactly "Inf" because "pi" is a float with limited precision
```

Python 3.x 3.5

```
math.atan(math.inf)
# Out: 1.5707963267948966 # This is just "pi / 2"
```

```
math.atan(float('inf'))
# Out: 1.5707963267948966 # This is just "pi / 2"
```

En dehors du `math.atan` il y a aussi une fonction `math.atan2` deux arguments, qui calcule le quadrant correct et évite les pièges de la division par zéro:

```
math.atan2(1, 2)   # Equivalent to "math.atan(1/2)"
# Out: 0.4636476090008061 # ≈ 26.57 degrees, 1st quadrant

math.atan2(-1, -2) # Not equal to "math.atan(-1/-2)" == "math.atan(1/2)"
# Out: -2.677945044588987 # ≈ -153.43 degrees (or 206.57 degrees), 3rd quadrant

math.atan2(1, 0)   # math.atan(1/0) would raise ZeroDivisionError
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Sinus hyperbolique, cosinus et tangente

```
# Hyperbolic sine function
math.sinh(math.pi) # = 11.548739357257746
math.asinh(1)      # = 0.8813735870195429

# Hyperbolic cosine function
math.cosh(math.pi) # = 11.591953275521519
math.acosh(1)      # = 0.0

# Hyperbolic tangent function
math.tanh(math.pi) # = 0.99627207622075
math.atanh(0.5)    # = 0.5493061443340549
```

Les constantes

modules `math` comprend deux constantes mathématiques couramment utilisées.

- `math.pi` - La constante mathématique pi
- `math.e` - La constante mathématique *e* (base du logarithme naturel)

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>>
```

Python 3.5 et supérieur ont des constantes pour l'infini et NaN ("pas un nombre"). L'ancienne syntaxe du passage d'une chaîne à `float()` fonctionne toujours.

Python 3.x 3.5

```
math.inf == float('inf')
# Out: True

-math.inf == float('-inf')
# Out: True

# NaN never compares equal to anything, even itself
math.nan == float('nan')
# Out: False
```

Nombres Imaginaires

Les nombres imaginaires en Python sont représentés par un "j" ou un "J" suivant le numéro cible.

```
1j         # Equivalent to the square root of -1.
1j * 1j    # = (-1+0j)
```

Infinity et NaN ("pas un nombre")

Dans toutes les versions de Python, nous pouvons représenter l'infini et NaN ("pas un nombre") comme suit:

```
pos_inf = float('inf')     # positive infinity
neg_inf = float('-inf')    # negative infinity
not_a_num = float('nan')   # NaN ("not a number")
```

En Python 3.5 et `math.nan` ultérieures, nous pouvons également utiliser les constantes définies `math.inf` et `math.nan` :

Python 3.x 3.5

```
pos_inf = math.inf
neg_inf = -math.inf
not_a_num = math.nan
```

Les représentations de chaîne s'affichent sous la forme `inf` et `-inf` et `nan` :

```
pos_inf, neg_inf, not_a_num
# Out: (inf, -inf, nan)
```

Nous pouvons tester l'infini positif ou négatif avec la méthode `isinf` :

```
math.isinf(pos_inf)
# Out: True

math.isinf(neg_inf)
# Out: True
```

Nous pouvons tester spécifiquement l'infini positif ou l'infini négatif par comparaison directe:

```
pos_inf == float('inf')    # or  == math.inf in Python 3.5+
# Out: True

neg_inf == float('-inf')   # or  == -math.inf in Python 3.5+
# Out: True

neg_inf == pos_inf
# Out: False
```

Python 3.2 et les versions ultérieures permettent également de vérifier la finitude:

Python 3.x 3.2

```
math.isfinite(pos_inf)
# Out: False

math.isfinite(0.0)
# Out: True
```

Les opérateurs de comparaison travaillent comme prévu pour l'infini positif et négatif:

```
import sys

sys.float_info.max
# Out: 1.7976931348623157e+308  (this is system-dependent)

pos_inf > sys.float_info.max
# Out: True

neg_inf < -sys.float_info.max
# Out: True
```

Mais si une expression arithmétique produit une valeur supérieure au maximum pouvant être représenté par un `float` , elle deviendra infinie:

```
pos_inf == sys.float_info.max * 1.0000001
# Out: True

neg_inf == -sys.float_info.max * 1.0000001
# Out: True
```

Cependant, la division par zéro ne donne pas un résultat d'infini (ou d'infini négatif le cas échéant), mais plutôt une exception `ZeroDivisionError` .

```
try:
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Division by zero")

# Out: Division by zero
```

Les opérations arithmétiques sur l'infini donnent simplement des résultats infinis, ou parfois NaN:

```
-5.0 * pos_inf == neg_inf
# Out: True

-5.0 * neg_inf == pos_inf
# Out: True

pos_inf * neg_inf == neg_inf
# Out: True

0.0 * pos_inf
# Out: nan

0.0 * neg_inf
# Out: nan

pos_inf / pos_inf
# Out: nan
```

NaN n'est jamais égal à rien, pas même à lui-même. Nous pouvons tester car c'est avec la méthode `isnan` :

```
not_a_num == not_a_num
# Out: False

math.isnan(not_a_num)
Out: True
```

NaN se compare toujours comme "non égal", mais jamais inférieur ou supérieur à:

```
not_a_num != 5.0   # or any random value
# Out: True

not_a_num > 5.0   or   not_a_num < 5.0   or   not_a_num == 5.0
# Out: False
```

Les opérations arithmétiques sur NaN donnent toujours NaN. Ceci inclut la multiplication par -1: il n'y a pas de "NaN négatif".

```
5.0 * not_a_num
# Out: nan

float('-nan')
# Out: nan
```

Python 3.x 3.5

```
-math.nan
# Out: nan
```

Il existe une différence subtile entre les anciennes versions `float` de NaN et infinity et les constantes de la bibliothèque `math` Python 3.5+:

Python 3.x 3.5

```
math.inf is math.inf, math.nan is math.nan
# Out: (True, True)

float('inf') is float('inf'), float('nan') is float('nan')
# Out: (False, False)
```

Pow pour une exponentiation plus rapide

En utilisant le module timeit depuis la ligne de commande:

```
> python -m timeit 'for x in xrange(50000): b = x**3'
10 loops, best of 3: 51.2 msec per loop
> python -m timeit 'from math import pow' 'for x in xrange(50000): b = pow(x,3)'
100 loops, best of 3: 9.15 msec per loop
```

L'opérateur `**` intégré est souvent utile, mais si les performances sont essentielles, utilisez math.pow. Veillez toutefois à noter que pow renvoie des flottants, même si les arguments sont des entiers:

```
> from math import pow
> pow(5,5)
3125.0
```

Les nombres complexes et le module cmath

Le module `cmath` est similaire au module `math` , mais définit les fonctions de manière appropriée pour le plan complexe.

Tout d'abord, les nombres complexes sont un type numérique qui fait partie du langage Python lui-même plutôt que d'être fourni par une classe de bibliothèque. Il n'est donc pas nécessaire d' `import cmath` pour les expressions arithmétiques ordinaires.

Notez que nous utilisons `j` (ou `J`) et non `i` .

```
z = 1 + 3j
```

Nous devons utiliser `1j` car `j` serait le nom d'une variable plutôt qu'un littéral numérique.

```
1j * 1j
Out: (-1+0j)

1j ** 1j
```

```
# Out: (0.20787957635076193+0j)     # "i to the i"  ==  math.e ** -(math.pi/2)
```

Nous avons la partie `real` et la partie `imag` (imaginaire), ainsi que le `conjugate` complexe:

```
# real part and imaginary part are both float type
z.real, z.imag
# Out: (1.0, 3.0)

z.conjugate()
# Out: (1-3j)    # z.conjugate() == z.real - z.imag * 1j
```

Les fonctions intégrées `abs` et `complex` font également partie du langage lui-même et ne nécessitent aucune importation:

```
abs(1 + 1j)
# Out: 1.4142135623730951     # square root of 2

complex(1)
# Out: (1+0j)

complex(imag=1)
# Out: (1j)

complex(1, 1)
# Out: (1+1j)
```

La fonction `complex` peut prendre une chaîne, mais elle ne peut pas avoir d'espaces:

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
# Exception: ValueError: complex() arg is a malformed string
```

Mais pour la plupart des fonctions, nous avons besoin du module, par exemple `sqrt` :

```
import cmath

cmath.sqrt(-1)
# Out: 1j
```

Naturellement, le comportement de `sqrt` est différent pour les nombres complexes et les nombres réels. Dans les `math` non complexes, la racine carrée d'un nombre négatif déclenche une exception:

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

Les fonctions sont fournies pour convertir vers et à partir des coordonnées polaires:

```
cmath.polar(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # == (sqrt(1 + 1), atan2(1, 1))

abs(1 + 1j), cmath.phase(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # same as previous calculation

cmath.rect(math.sqrt(2), math.atan(1))
# Out: (1.0000000000000002+1.0000000000000002j)
```

Le champ mathématique de l'analyse complexe dépasse le cadre de cet exemple, mais de nombreuses fonctions dans le plan complexe ont une "coupure de branche", généralement le long de l'axe réel ou de l'axe imaginaire. La plupart des plates-formes modernes prennent en charge le «zéro signé» tel que spécifié dans la norme IEEE 754, qui assure la continuité de ces fonctions des deux côtés de la découpe. L'exemple suivant provient de la documentation Python:

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
# Out: -3.141592653589793
```

Le module `cmath` fournit également de nombreuses fonctions avec des contreparties directes du module `math` .

En plus de `sqrt` , il existe des versions complexes de `exp` , `log` , `log10` , les fonctions trigonométriques et leurs inverses (`sin` , `cos` , `tan` , `asin` , `acos` , `atan`), et les fonctions hyperboliques et leurs inverses (`sinh` , `cosh` , `tanh` , `asinh` , `acosh` , `atanh`). Notez cependant qu'il n'y a pas d'homologue complexe de `math.atan2` , la forme à deux arguments de l'arctangent.

```
cmath.log(1+1j)
# Out: (0.34657359027997264+0.7853981633974483j)

cmath.exp(1j * cmath.pi)
# Out: (-1+1.2246467991473532e-16j)   # e to the i pi == -1, within rounding error
```

Les constantes `pi` et `e` sont fournies. Notez que ceux-ci sont `float` et non `complex` .

```
type(cmath.pi)
# Out: <class 'float'>
```

Le module `cmath` fournit également des versions complexes de `isinf` , et (pour Python 3.2+) `isfinite` . Voir " Infinity et NaN ". Un nombre complexe est considéré comme infini si sa partie réelle ou sa partie imaginaire est infinie.

```
cmath.isinf(complex(float('inf'), 0.0))
# Out: True
```

De même, le module `cmath` fournit une version complexe de `isnan` . Voir " Infinity et NaN ". Un nombre complexe est considéré "pas un nombre" si sa partie réelle ou sa partie imaginaire est "pas un nombre".

```
cmath.isnan(0.0, float('nan'))
# Out: True
```

Notez qu'il n'y a pas `cmath` contrepartie des `math.inf` et `math.nan` constantes (de Python 3.5 et versions ultérieures)

Python 3.x 3.5

```
cmath.isinf(complex(0.0, math.inf))
# Out: True

cmath.isnan(complex(math.nan, 0.0))
# Out: True

cmath.inf
# Exception: AttributeError: module 'cmath' has no attribute 'inf'
```

Dans Python 3.5 et `isclose` ultérieures, il existe une méthode `isclose` dans les modules `cmath` et `math` .

Python 3.x 3.5

```
z = cmath.rect(*cmath.polar(1+1j))

z
# Out: (1.0000000000000002+1.0000000000000002j)

cmath.isclose(z, 1+1j)
# True
```

Lire Module Math en ligne: https://riptutorial.com/fr/python/topic/230/module-math

Chapitre 128: Module opérateur

Examples

Opérateurs comme alternative à un opérateur infixe

Pour chaque opérateur infixe, par exemple `+` il y a une fonction `operator` (`operator.add` for `+`):

```
1 + 1
# Output: 2
from operator import add
add(1, 1)
# Output: 2
```

même si la documentation principale indique que pour les opérateurs arithmétiques, seule la saisie numérique est autorisée, il *est* possible:

```
from operator import mul
mul('a', 10)
# Output: 'aaaaaaaaaa'
mul([3], 3)
# Output: [3, 3, 3]
```

Voir aussi: mappage de l'opération à la fonction opérateur dans la documentation officielle de Python .

Méthode

Au lieu de cette fonction `lambda` qui appelle la méthode explicitement:

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist))     # Keep only elements that start with 'd'
# Output: ['duck']
```

on pourrait utiliser une fonction opérateur qui fait la même chose:

```
from operator import methodcaller
list(filter(methodcaller('startswith', 'd'), alist)) # Does the same but is faster.
# Output: ['duck']
```

Itemgetter

Regroupement des paires clé-valeur d'un dictionnaire par la valeur avec `itemgetter` :

```
from itertools import groupby
from operator import itemgetter
adict = {'a': 1, 'b': 5, 'c': 1}
```

```
dict((i, dict(v)) for i, v in groupby(adict.items(), itemgetter(1)))
# Output: {1: {'a': 1, 'c': 1}, 5: {'b': 5}}
```

ce qui est équivalent (mais plus rapide) à une fonction `lambda` comme ceci:

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

Ou trier une liste de tuples par le second élément en premier lieu le premier élément en tant que secondaire:

```
alist_of_tuples = [(5,2), (1,3), (2,2)]
sorted(alist_of_tuples, key=itemgetter(1,0))
# Output: [(2, 2), (5, 2), (1, 3)]
```

Lire Module opérateur en ligne: https://riptutorial.com/fr/python/topic/257/module-operateur

Chapitre 129: module pyautogui

Introduction

pyautogui est un module utilisé pour contrôler la souris et le clavier. Ce module est essentiellement utilisé pour automatiser les tâches de clic de souris et de clavier. Pour la souris, les coordonnées de l'écran (0,0) partent du coin supérieur gauche. Si vous êtes hors de contrôle, déplacez rapidement le curseur de la souris vers le haut à gauche, cela prendra le contrôle de la souris et du clavier du Python et vous le rendra.

Examples

Fonctions de la souris

Ce sont certaines des fonctions de souris utiles pour contrôler la souris.

```
size()           #gave you the size of the screen
position()     #return current position of mouse
moveTo(200,0,duration=1.5)     #move the cursor  to (200,0) position  with 1.5 second delay

moveRel()          #move the cursor relative to your current position.
click(337,46)           #it will click on the position mention there
dragRel()              #it will drag the mouse relative to position
pyautogui.displayMousePosition()     #gave you the current mouse position but should be done
on terminal.
```

Fonctions du clavier

Ce sont quelques fonctions utiles du clavier pour automatiser l’appui sur les touches.

```
typewrite('')    #this will type the string on the screen where current window has focused.
typewrite(['a','b','left','left','X','Y'])
pyautogui.KEYBOARD_KEYS    #get the list of all the keyboard_keys.
pyautogui.hotkey('ctrl','o')    #for the combination of keys to enter.
```

Capture d'écran et reconnaissance d'image

Ces fonctions vous aideront à prendre la capture d'écran et à faire correspondre l'image à la partie de l'écran.

```
.screenshot('c:\\path')        #get the screenshot.
.locateOnScreen('c:\\path')    #search that image on screen and get the coordinates for you.
locateCenterOnScreen('c:\\path')       #get the coordinate for the image on screen.
```

Lire module pyautogui en ligne: https://riptutorial.com/fr/python/topic/9432/module-pyautogui

Chapitre 130: Module Sqlite3

Examples

Sqlite3 - Ne nécessite pas de processus serveur séparé.

Le module sqlite3 a été écrit par Gerhard Häring. Pour utiliser le module, vous devez d'abord créer un objet Connection qui représente la base de données. Ici, les données seront stockées dans le fichier example.db:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

Vous pouvez également fournir le nom spécial: memory: pour créer une base de données en RAM. Une fois que vous avez une connexion, vous pouvez créer un objet Cursor et appeler sa méthode execute () pour exécuter les commandes SQL:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
         (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

Obtenir les valeurs de la base de données et la gestion des erreurs

Récupération des valeurs de la base de données SQLite3.

Imprimer les valeurs de ligne renvoyées par la requête select

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute("SELECT * from table_name where id=cust_id")
for row in c:
    print row # will be a list
```

Pour récupérer la méthode unique correspondant à fetchone ()

```
print c.fetchone()
```

Pour plusieurs lignes, utilisez la méthode fetchall ()

```
a=c.fetchall() #which is similar to list(cursor) method used previously
for row in a:
    print row
```

La gestion des erreurs peut être effectuée à l'aide de la fonction intégrée sqlite3.Error

```
try:
    #SQL Code
except sqlite3.Error as e:
    print "An error occurred:", e.args[0]
```

Lire Module Sqlite3 en ligne: https://riptutorial.com/fr/python/topic/7754/module-sqlite3

Chapitre 131: Module Webbrowser

Introduction

Selon la documentation standard de Python, le module Webbrowser fournit une interface de haut niveau pour permettre l'affichage des documents Web aux utilisateurs. Cette rubrique explique et démontre l'utilisation correcte du module navigateur Web.

Syntaxe

- `webbrowser.open(url, new=0, autoraise=False)`
- `webbrowser.open_new(url)`
- `webbrowser.open_new_tab(url)`
- `webbrowser.get(usage=None)`
- `webbrowser.register(name, constructor, instance=None)`

Paramètres

Paramètre	Détails
`webbrowser.open()`	
URL	l'URL à ouvrir dans le navigateur Web
Nouveau	0 ouvre l'URL dans l'onglet existant, 1 s'ouvre dans une nouvelle fenêtre, 2 s'ouvre dans un nouvel onglet
autoraise	Si défini sur True, la fenêtre sera déplacée au-dessus des autres fenêtres
`webbrowser.open_new()`	
URL	l'URL à ouvrir dans le navigateur Web
`webbrowser.open_new_tab()`	
URL	l'URL à ouvrir dans le navigateur Web
`webbrowser.get()`	
en utilisant	le navigateur à utiliser
`webbrowser.register()`	
URL	nom du navigateur
constructeur	chemin d'accès au navigateur exécutable (aide)
exemple	Une instance d'un navigateur Web renvoyée par la méthode

Paramètre	Détails
	`webbrowser.get()`

Remarques

Le tableau suivant répertorie les types de navigateur prédéfinis. La colonne de gauche contient des noms pouvant être transmis à la méthode `webbrowser.get()` et la colonne de droite répertorie les noms de classe pour chaque type de navigateur.

Nom du type	Nom du cours
`'mozilla'`	`Mozilla('mozilla')`
`'firefox'`	`Mozilla('mozilla')`
`'netscape'`	`Mozilla('netscape')`
`'galeon'`	`Galeon('galeon')`
`'epiphany'`	`Galeon('epiphany')`
`'skipstone'`	`BackgroundBrowser('skipstone')`
`'kfmclient'`	`Konqueror()`
`'konqueror'`	`Konqueror()`
`'kfm'`	`Konqueror()`
`'mosaic'`	`BackgroundBrowser('mosaic')`
`'opera'`	`Opera()`
`'grail'`	`Grail()`
`'links'`	`GenericBrowser('links')`
`'elinks'`	`Elinks('elinks')`
`'lynx'`	`GenericBrowser('lynx')`
`'w3m'`	`GenericBrowser('w3m')`
`'windows-default'`	`WindowsDefault`
`'macosx'`	`MacOSX('default')`
`'safari'`	`MacOSX('safari')`
`'google-chrome'`	`Chrome('google-chrome')`
`'chrome'`	`Chrome('chrome')`
`'chromium'`	`Chromium('chromium')`
`'chromium-browser'`	`Chromium('chromium-browser')`

Examples

Ouverture d'une URL avec le navigateur par défaut

Pour simplement ouvrir une URL, utilisez la méthode `webbrowser.open()` :

```
import webbrowser
webbrowser.open("http://stackoverflow.com")
```

Si une fenêtre de navigateur est actuellement ouverte, la méthode ouvre un nouvel onglet à l'URL spécifiée. Si aucune fenêtre n'est ouverte, la méthode ouvre le navigateur par défaut du système d'exploitation et accède à l'URL du paramètre. La méthode ouverte prend en charge les paramètres suivants:

- `url` - l'URL à ouvrir dans le navigateur Web (chaîne) **[requis]**
- `new` - 0 ouvre dans l'onglet existant, 1 ouvre une nouvelle fenêtre, 2 ouvre un nouvel onglet (entier) **[par défaut 0]**
- `autoraise` - si défini sur True, la fenêtre sera déplacée au-dessus des fenêtres des autres applications (booléenne) **[False par défaut]**

Notez que les arguments `new` et `autoraise` fonctionnent rarement car la majorité des navigateurs modernes refusent ces commandes.

Webbrowser peut également essayer d'ouvrir des URL dans de nouvelles fenêtres avec la méthode `open_new` :

```
import webbrowser
webbrowser.open_new("http://stackoverflow.com")
```

Cette méthode est généralement ignorée par les navigateurs modernes et l'URL est généralement ouverte dans un nouvel onglet. L'ouverture d'un nouvel onglet peut être tentée par le module en utilisant la méthode `open_new_tab` :

```
import webbrowser
webbrowser.open_new_tab("http://stackoverflow.com")
```

Ouverture d'une URL avec différents navigateurs

Le module webbrowser prend également en charge différents navigateurs utilisant les méthodes `register()` et `get()` . La méthode get est utilisée pour créer un contrôleur de navigateur en utilisant le chemin d'accès d'un exécutable spécifique et la méthode register est utilisée pour attacher ces exécutables à des types de navigateur prédéfinis pour une utilisation future, généralement lorsque plusieurs types de navigateurs sont utilisés.

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

Enregistrer un type de navigateur:

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
webbrowser.register('firefox', None, ff)
# Now to refer to use Firefox in the future you can use this
webbrowser.get('firefox').open("https://stackoverflow.com/")
```

Lire Module Webbrowser en ligne: https://riptutorial.com/fr/python/topic/8676/module-webbrowser

Chapitre 132: Multithreading

Introduction

Les threads permettent aux programmes Python de gérer plusieurs fonctions à la fois, plutôt que d'exécuter une séquence de commandes individuellement. Cette rubrique explique les principes qui sous-tendent les threads et illustre son utilisation.

Examples

Bases du multithreading

En utilisant le module de `threading` , un nouveau thread d'exécution peut être démarré en créant un nouveau `threading.Thread` et en lui assignant une fonction à exécuter:

```
import threading

def foo():
  print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

Le paramètre `target` référence à la fonction (ou à l'objet appelable) à exécuter. Le thread ne commencera pas l'exécution tant que l'objet `Thread` n'a pas `start` .

Commencer un fil

```
my_thread.start() # prints 'Hello threading!'
```

Maintenant que `my_thread` a été exécuté et terminé, l'appel `start` nouveau produira une `RuntimeError` . Si vous souhaitez exécuter votre thread en tant que démon, en passant le `daemon=True` kwarg ou en définissant `my_thread.daemon` sur `True` avant d'appeler `start()` , votre `Thread` s'exécute en arrière-plan en tant que démon.

Rejoindre un fil

Dans les cas où vous divisez un gros job en plusieurs petits et que vous souhaitez les exécuter simultanément, mais que vous devez attendre qu'ils soient tous terminés avant de continuer, `Thread.join()` est la méthode que vous recherchez.

Par exemple, supposons que vous souhaitiez télécharger plusieurs pages d'un site Web et les compiler en une seule page. Vous feriez ceci:

```
import requests
from threading import Thread
from queue import Queue
```

```
q = Queue(maxsize=20)
def put_page_to_q(page_num):
    q.put(requests.get('http://some-website.com/page_%s.html' % page_num)

def compile(q):
    # magic function that needs all pages before being able to be executed
    if not q.full():
        raise ValueError
    else:
        print("Done compiling!")

threads = []
for page_num in range(20):
     t = Thread(target=requests.get, args=(page_num,))
     t.start()
     threads.append(t)

# Next, join all threads to make sure all threads are done running before
# we continue. join() is a blocking call (unless specified otherwise using
# the kwarg blocking=False when calling join)
for t in threads:
    t.join()

# Call compile() now, since all threads have completed
compile(q)
```

Un aperçu plus précis de la manière dont fonctionne `join()` peut être trouvé ici .

Créer une classe de threads personnalisée

En utilisant la classe `threading.Thread` , on peut sous-classer la nouvelle classe de threads personnalisée. nous devons remplacer la méthode d' `run` dans une sous-classe.

```
from threading import Thread
import time

class Sleepy(Thread):

    def run(self):
        time.sleep(5)
        print("Hello form Thread")

if __name__ == "__main__":
    t = Sleepy()
    t.start()      # start method automatic call Thread class run method.
    # print 'The main program continues to run in foreground.'
    t.join()
    print("The main program continues to run in the foreground.")
```

Communiquer entre les threads

Il y a plusieurs threads dans votre code et vous devez communiquer entre eux en toute sécurité.

Vous pouvez utiliser une `Queue` d' `Queue` de la bibliothèque de `queue` .

```
from queue import Queue
```

```
from threading import Thread

# create a data producer
def producer(output_queue):
    while True:
        data = data_computation()

        output_queue.put(data)

# create a consumer
def consumer(input_queue):
    while True:
        # retrieve data (blocking)
        data = input_queue.get()

        # do something with the data

        # indicate data has been consumed
        input_queue.task_done()
```

Création de threads producteur et consommateur avec une file d'attente partagée

```
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Création d'un pool de travailleurs

Utilisation de `threading` & `queue` :

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_server(addr, nworkers):
    print('Echo server running at', addr)
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

En utilisant `concurrent.futures.Threadpoolexecutor` :

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_server(addr):
    print('Echo server running at', addr)
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

Python Cookbook, 3ème édition, par David Beazley et Brian K. Jones (O'Reilly). Copyright 2013 David Beazley et Brian Jones, 978-1-449-34037-7.

Utilisation avancée de multithreads

Cette section contient certains des exemples les plus avancés réalisés avec Multithreading.

Imprimante avancée (enregistreur)

Un thread qui imprime tout est reçu et modifie la sortie en fonction de la largeur du terminal. Le point intéressant est que la sortie "déjà écrite" est modifiée lorsque la largeur du terminal change.

```
#!/usr/bin/env python2

import threading
import Queue
import time
import sys
import subprocess
from backports.shutil_get_terminal_size import get_terminal_size

printq = Queue.Queue()
interrupt = False
lines = []

def main():

    ptt = threading.Thread(target=printer) # Turn the printer on
    ptt.daemon = True
    ptt.start()

    # Stupid example of stuff to print
    for i in xrange(1,100):
        printq.put(' '.join([str(x) for x in range(1,i)]))           # The actual way to send
stuff to the printer
        time.sleep(.5)

def split_line(line, cols):
    if len(line) > cols:
        new_line = ''
        ww = line.split()
```

```
        i = 0
        while len(new_line) <= (cols - len(ww[i]) - 1):
            new_line += ww[i] + ' '
            i += 1
            print len(new_line)
        if new_line == '':
            return (line, '')

        return (new_line, ' '.join(ww[i:]))
    else:
        return (line, '')


def printer():

    while True:
        cols, rows = get_terminal_size() # Get the terminal dimensions
        msg = '#' + '-' * (cols - 2) + '#\n' # Create the
        try:
            new_line = str(printq.get_nowait())
            if new_line != '!@#EXIT#@!': # A nice way to turn the printer
                                         # thread out gracefully
                lines.append(new_line)
                printq.task_done()
            else:
                printq.task_done()
                sys.exit()
        except Queue.Empty:
            pass

        # Build the new message to show and split too long lines
        for line in lines:
            res = line          # The following is to split lines which are
                                # longer than cols.
            while len(res) !=0:
                toprint, res = split_line(res, cols)
                msg += '\n' + toprint

        # Clear the shell and print the new output
        subprocess.check_call('clear') # Keep the shell clean
        sys.stdout.write(msg)
        sys.stdout.flush()
        time.sleep(.5)
```

Thread bloquable avec une boucle while

```
import threading
import time

class StoppableThread(threading.Thread):
    """Thread class with a stop() method. The thread itself has to check
    regularly for the stopped() condition."""

    def __init__(self):
        super(StoppableThread, self).__init__()
        self._stop_event = threading.Event()

    def stop(self):
        self._stop_event.set()
```

```
    def join(self, *args, **kwargs):
        self.stop()
        super(StoppableThread,self).join(*args, **kwargs)

    def run()
        while not self._stop_event.is_set():
            print("Still running!")
            time.sleep(2)
        print("stopped!"
```

Basé sur cette question .

Lire Multithreading en ligne: https://riptutorial.com/fr/python/topic/544/multithreading

Chapitre 133: Multitraitement

Examples

Exécution de deux processus simples

Un exemple simple d'utilisation de plusieurs processus serait deux processus (travailleurs) exécutés séparément. Dans l'exemple suivant, deux processus sont lancés:

- `countUp()` compte 1, chaque seconde.
- `countDown()` compte 1, chaque seconde.

```
import multiprocessing
import time
from random import randint

def countUp():
    i = 0
    while i <= 3:
        print('Up:\t{}'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i += 1

def countDown():
    i = 3
    while i >= 0:
        print('Down:\t{}'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i -= 1

if __name__ == '__main__':
    # Initiate the workers.
    workerUp = multiprocessing.Process(target=countUp)
    workerDown = multiprocessing.Process(target=countDown)

    # Start the workers.
    workerUp.start()
    workerDown.start()

    # Join the workers. This will block in the main (parent) process
    # until the workers are complete.
    workerUp.join()
    workerDown.join()
```

Le résultat est le suivant:

```
Up:    0
Down:    3
Up:    1
Up:    2
Down:    2
Up:    3
Down:    1
Down:    0
```

Utilisation du pool et de la carte

```
from multiprocessing import Pool

def cube(x):
    return x ** 3

if __name__ == "__main__":
    pool = Pool(5)
    result = pool.map(cube, [0, 1, 2, 3])
```

`Pool` est une classe qui gère plusieurs `Workers` (processus) en coulisse et vous permet, au programmeur, de l'utiliser.

`Pool(5)` crée un nouveau pool avec 5 processus, et `pool.map` fonctionne comme Map mais utilise plusieurs processus (la quantité définie lors de la création du pool).

Des résultats similaires peuvent être obtenus en utilisant `map_async` , `apply` et `apply_async` qui peuvent être trouvés dans la documentation .

Lire Multitraitement en ligne: https://riptutorial.com/fr/python/topic/3601/multitraitement

Chapitre 134: Mutable vs immuable (et lavable) en Python

Examples

Mutable vs immuable

Il existe deux types de types dans Python. Types immuables et types mutables.

Immuables

Un objet d'un type immuable ne peut pas être modifié. Toute tentative de modification de l'objet entraînera la création d'une copie.

Cette catégorie comprend: les entiers, les flottants, les complexes, les chaînes, les octets, les tuples, les plages et les frozensets.

Pour mettre en évidence cette propriété, jouons avec l' `id` builtin. Cette fonction renvoie l'identificateur unique de l'objet passé en paramètre. Si l'identifiant est le même, c'est le même objet. Si cela change, alors c'est un autre objet. *(Certains disent que c'est en fait l'adresse mémoire de l'objet, mais méfiez-vous d'eux, ils sont du côté obscur de la force ...)*

```
>>> a = 1
>>> id(a)
140128142243264
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

Ok, 1 n'est pas 3 ... Dernières nouvelles ... Peut-être pas. Cependant, ce comportement est souvent oublié en ce qui concerne les types plus complexes, en particulier les chaînes de caractères.

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
140128123955504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

Aha! Voir? Nous pouvons le modifier!

```
>>> id(stack)
140128123911472
```

Bien qu'il semble que nous puissions changer la chaîne nommée par la `stack` variables, ce que nous faisons réellement, c'est de créer un nouvel objet pour contenir le résultat de la concaténation. Nous sommes dupes parce que dans le processus, le vieil objet ne va nulle part, donc il est détruit. Dans une autre situation, cela aurait été plus évident:

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

Dans ce cas, il est clair que si nous voulons conserver la première chaîne, nous avons besoin d'une copie. Mais est-ce si évident pour d'autres types?

Exercice

Maintenant, sachant comment fonctionnent les types immuables, que diriez-vous avec le morceau de code ci-dessous? Est-ce sage?

```
s = ""
for i in range(1, 1000):
    s += str(i)
    s += ","
```

Mutables

Un objet de type mutable peut être modifié et modifié *in situ* . Aucune copie implicite n'est faite.

Cette catégorie comprend: les listes, les dictionnaires, les séries et les ensembles.

Continuons à jouer avec notre petite fonction d' `id` .

```
>>> b = bytearray(b'Stack')
>>> b
bytearray(b'Stack')
>>> b = bytearray(b'Stack')
>>> id(b)
140128030688288
>>> b += b'Overflow'
>>> b
bytearray(b'StackOverflow')
>>> id(b)
140128030688288
```

(En remarque, j'utilise des octets contenant des données ascii pour clarifier mon propos, mais souvenez-vous que les octets ne sont pas conçus pour contenir des données textuelles. La force

peut me pardonner.)

Qu'avons-nous? Nous créons un bytearray, le modifions et en utilisant l' `id` , nous pouvons nous assurer qu'il s'agit du même objet, modifié. Pas une copie

Bien sûr, si un objet doit être modifié souvent, un type mutable fait un travail bien meilleur qu'un type immuable. Malheureusement, la réalité de cette propriété est souvent oubliée lorsqu'elle fait le plus mal.

```
>>> c = b
>>> c += b' rocks!'
>>> c
bytearray(b'StackOverflow rocks!')
```

D'accord...

```
>>> b
bytearray(b'StackOverflow rocks!')
```

Waiiit une seconde ...

```
>>> id(c) == id(b)
True
```

Effectivement. `c` n'est pas une copie de `b` . `c` est `b` .

Exercice

Maintenant, vous comprenez mieux quel effet secondaire un type mutable implique, pouvez-vous expliquer ce qui ne va pas dans cet exemple?

```
>>> ll = [ [] ]*4 # Create a list of 4 lists to contain our results
>>> ll
[[], [], [], []]
>>> ll[0].append(23) # Add result 23 to first list
>>> ll
[[23], [23], [23], [23]]
>>> # Oops...
```

Mutable et immuable comme arguments

L'un des principaux cas d'utilisation lorsqu'un développeur doit prendre en compte la mutabilité est lorsqu'il transmet des arguments à une fonction. Ceci est très important, car cela déterminera la capacité de la fonction à modifier des objets qui n'appartiennent pas à sa portée, c'est-à-dire si la fonction a des effets secondaires. Ceci est également important pour comprendre où le résultat d'une fonction doit être rendu disponible.

```
>>> def list_add3(lin):
    lin += [3]
```

```
    return lin

>>> a = [1, 2, 3]
>>> b = list_add3(a)
>>> b
[1, 2, 3, 3]
>>> a
[1, 2, 3, 3]
```

Ici, l'erreur est de penser que `lin` , en tant que paramètre de la fonction, peut être modifié localement. Au lieu de cela, `lin` et `a` référence le même objet. Comme cet objet est mutable, la modification est effectuée sur place, ce qui signifie que l'objet référencé à la fois par `lin` et `a` est modifié. `lin` n'a pas vraiment besoin d'être retourné, car nous avons déjà une référence à cet objet sous la forme d' `a` . `a` et `b` finissent par référencer le même objet.

Cela ne va pas la même chose pour les tuples.

```
>>> def tuple_add3(tin):
    tin += (3,)
    return tin

>>> a = (1, 2, 3)
>>> b = tuple_add3(a)
>>> b
(1, 2, 3, 3)
>>> a
(1, 2, 3)
```

Au début de la fonction, `tin` et `a` référence le même objet. Mais c'est un objet immuable. Ainsi , lorsque la fonction tente de le modifier, l' `tin` recevoir un nouvel objet avec la modification, en `a` conserve une référence à l'objet d' origine. Dans ce cas, le retour de l' `tin` est obligatoire ou le nouvel objet serait perdu.

Exercice

```
>>> def yoda(prologue, sentence):
    sentence.reverse()
    prologue += " ".join(sentence)
    return prologue

>>> focused = ["You must", "stay focused"]
>>> saying = "Yoda said: "
>>> yoda_sentence = yoda(saying, focused)
```

Note: `reverse` fonctionne sur place.

Que pensez-vous de cette fonction? At-il des effets secondaires? Le retour est-il nécessaire? Après l'appel, quelle est la valeur de `saying` ? De `focused` ? Que se passe-t-il si la fonction est appelée à nouveau avec les mêmes paramètres?

Lire Mutable vs immuable (et lavable) en Python en ligne:
https://riptutorial.com/fr/python/topic/9182/mutable-vs-immuable--et-lavable--en-python

Chapitre 135: Neo4j et Cypher utilisant Py2Neo

Examples

Importation et authentification

```
from py2neo import authenticate, Graph, Node, Relationship
authenticate("localhost:7474", "neo4j", "<pass>")
graph = Graph()
```

Vous devez vous assurer que votre base de données Neo4j existe sur localhost: 7474 avec les informations d'identification appropriées.

L'objet `graph` est votre interface avec l'instance neo4j dans le reste de votre code python. Merci de faire de cette variable une variable globale, vous devriez la garder dans la méthode `__init__` une classe.

Ajout de nœuds au graphique Neo4j

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    article.properties["title"] = results[r]['news_title']
    article.properties["timestamp"] = results[r]['news_timestamp']
    article.push()
    [...]
```

L'ajout de nœuds au graphique est assez simple, `graph.merge_one` est important car il évite les doublons. (Si vous exécutez le script deux fois, la deuxième fois, il mettra à jour le titre et ne créera pas de nouveaux nœuds pour les mêmes articles)

`timestamp` devrait être un entier et non une chaîne de date car neo4j n'a pas vraiment de type de données de date. Cela provoque des problèmes de tri lorsque vous stockez la date sous la forme '05 -06-1989 '

`article.push()` est l'appel qui valide effectivement l'opération dans neo4j. N'oubliez pas cette étape.

Ajout de relations au graphique Neo4j

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    if 'LOCATION' in results[r].keys():
        for loc in results[r]['LOCATION']:
            loc = graph.merge_one("Location", "name", loc)
```

```
            try:
                rel = graph.create_unique(Relationship(article, "about_place", loc))
            except Exception, e:
                print e
```

`create_unique` est important pour éviter les doublons. Mais sinon c'est une opération assez simple. Le nom de la relation est également important car vous l'utiliseriez dans les cas avancés.

Requête 1: saisie semi-automatique sur les titres d'actualités

```
def get_autocomplete(text):
    query = """
    start n = node(*) where n.name =~ '(?i)%s.*' return n.name,labels(n) limit 10;
    """
    query = query % (text)
    obj = []
    for res in graph.cypher.execute(query):
        # print res[0],res[1]
        obj.append({'name':res[0],'entity_type':res[1]})
    return res
```

Ceci est un exemple de requête cryptage pour obtenir tous les nœuds avec le `name` la propriété qui commence par le `text` l'argument.

Requête 2: Obtenir des articles par lieu à une date donnée

```
def search_news_by_entity(location,timestamp):
    query = """
    MATCH (n)-[]->(l)
    where l.name='%s' and n.timestamp='%s'
    RETURN n.news_id limit 10
    """

    query = query % (location,timestamp)

    news_ids = []
    for res in graph.cypher.execute(query):
        news_ids.append(str(res[0]))

    return news_ids
```

Vous pouvez utiliser cette requête pour rechercher tous les articles `(n)` connectés à un emplacement `(l)` par une relation.

Échantillons d'interrogation

Compter les articles connectés à une personne en particulier au fil du temps

```
MATCH (n)-[]->(l)
where l.name='Donald Trump'
RETURN n.date,count(*) order by n.date
```

Recherchez d'autres personnes / emplacements connectés aux mêmes articles d'actualité que

Trump avec au moins 5 nœuds de relation au total.

```
MATCH (n:NewsArticle)-[]->(l)
where l.name='Donald Trump'
MATCH (n:NewsArticle)-[]->(m)
with m,count(n) as num where num>5
return labels(m)[0],(m.name), num order by num desc limit 10
```

Lire Neo4j et Cypher utilisant Py2Neo en ligne: https://riptutorial.com/fr/python/topic/5841/neo4j-et-cypher-utilisant-py2neo

Chapitre 136: Noeud Liste liée

Examples

Écrire un nœud de liste lié simple en python

Une liste liée est soit:

- la liste vide, représentée par None ou
- un nœud contenant un objet cargo et une référence à une liste chaînée.

```
#! /usr/bin/env python

class Node:
      def __init__(self, cargo=None, next=None):
          self.car = cargo
          self.cdr = next
      def __str__(self):
          return str(self.car)


      def display(lst):
          if lst:
             w("%s " % lst)
             display(lst.cdr)
          else:
             w("nil\n")
```

Lire Noeud Liste liée en ligne: https://riptutorial.com/fr/python/topic/6916/noeud-liste-liee

Chapitre 137: Objets de propriété

Remarques

Remarque : dans Python 2, assurez-vous que votre classe hérite de l'objet (ce qui en fait une classe de style nouveau) afin que toutes les fonctionnalités des propriétés soient disponibles.

Examples

Utiliser le décorateur @property

Le décorateur `@property` peut être utilisé pour définir des méthodes dans une classe qui agissent comme des attributs. Un exemple où cela peut être utile est lorsque vous exposez des informations qui peuvent nécessiter une recherche initiale (coûteuse) et une récupération simple par la suite.

Étant donné un module `foobar.py` :

```
class Foo(object):
    def __init__(self):
        self.__bar = None

    @property
    def bar(self):
        if self.__bar is None:
            self.__bar = some_expensive_lookup_operation()
        return self.__bar
```

alors

```
>>> from foobar import Foo
>>> foo = Foo()
>>> print(foo.bar)  # This will take some time since bar is None after initialization
42
>>> print(foo.bar)  # This is much faster since bar has a value now
42
```

Utilisation du décorateur @property pour les propriétés en lecture-écriture

Si vous souhaitez utiliser `@property` pour implémenter un comportement personnalisé pour définir et obtenir, utilisez ce modèle:

```
class Cash(object):
    def __init__(self, value):
        self.value = value
    @property
    def formatted(self):
        return '${:.2f}'.format(self.value)
```

```
    @formatted.setter
    def formatted(self, new):
        self.value = float(new[1:])
```

Pour utiliser ceci:

```
>>> wallet = Cash(2.50)
>>> print(wallet.formatted)
$2.50
>>> print(wallet.value)
2.5
>>> wallet.formatted = '$123.45'
>>> print(wallet.formatted)
$123.45
>>> print(wallet.value)
123.45
```

Ne substituer qu'un getter, un setter ou un deleter d'un objet de propriété

Lorsque vous héritez d'une classe avec une propriété, vous pouvez fournir une nouvelle implémentation pour une ou plusieurs des fonctions `getter` , `setter` ou `deleter` , en référençant l'objet propriété *sur la classe parent* :

```
class BaseClass(object):
    @property
    def foo(self):
        return some_calculated_value()

    @foo.setter
    def foo(self, value):
        do_something_with_value(value)


class DerivedClass(BaseClass):
    @BaseClass.foo.setter
    def foo(self, value):
        do_something_different_with_value(value)
```

Vous pouvez également ajouter un setter ou un deleter là où il n'y en avait pas auparavant sur la classe de base.

Utiliser des propriétés sans décorateurs

Bien que l'utilisation de la syntaxe de décorateur (avec le @) soit pratique, elle cache aussi un peu. Vous pouvez utiliser des propriétés directement, sans décorateurs. L'exemple suivant de Python 3.x montre ceci:

```
class A:
    p = 1234
    def getX (self):
        return self._x

    def setX (self, value):
```

```
        self._x = value

    def getY (self):
        return self._y

    def setY (self, value):
        self._y = 1000 + value    # Weird but possible

    def getY2 (self):
        return self._y

    def setY2 (self, value):
        self._y = value

    def getT    (self):
        return self._t

    def setT (self, value):
        self._t = value

    def getU (self):
        return self._u + 10000

    def setU (self, value):
        self._u = value - 5000

    x, y, y2 = property (getX, setX), property (getY, setY), property (getY2, setY2)
    t = property (getT, setT)
    u = property (getU, setU)

A.q = 5678

class B:
    def getZ (self):
        return self.z_

    def setZ (self, value):
        self.z_ = value

    z = property (getZ, setZ)

class C:
    def __init__ (self):
        self.offset = 1234

    def getW (self):
        return self.w_ + self.offset

    def setW (self, value):
        self.w_ = value - self.offset

    w = property (getW, setW)

a1 = A ()
a2 = A ()

a1.y2 = 1000
a2.y2 = 2000

a1.x = 5
a1.y = 6
```

```
a2.x = 7
a2.y = 8

a1.t = 77
a1.u = 88

print (a1.x, a1.y, a1.y2)
print (a2.x, a2.y, a2.y2)
print (a1.p, a2.p, a1.q, a2.q)

print (a1.t, a1.u)

b = B ()
c = C ()

b.z = 100100
c.z = 200200
c.w = 300300

print (a1.x, b.z, c.z, c.w)

c.w = 400400
c.z = 500500
b.z = 600600

print (a1.x, b.z, c.z, c.w)
```

Lire Objets de propriété en ligne: https://riptutorial.com/fr/python/topic/2050/objets-de-propriete

Chapitre 138: Opérateurs booléens

Examples

et

Évalue le deuxième argument si et seulement si les deux arguments sont véridiques. Sinon, évalue au premier argument de falsey.

```
x = True
y = True
z = x and y # z = True

x = True
y = False
z = x and y # z = False

x = False
y = True
z = x and y # z = False

x = False
y = False
z = x and y # z = False

x = 1
y = 1
z = x and y # z = y, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 0
y = 1
z = x and y # z = x, so z = 0 (see above)

x = 1
y = 0
z = x and y # z = y, so z = 0 (see above)

x = 0
y = 0
z = x and y # z = x, so z = 0 (see above)
```

Les `1` dans l'exemple ci-dessus peuvent être changés en n'importe quelle valeur, et les `0` peuvent être changés en n'importe quelle valeur.

ou

Évalue le premier argument véridique si l'un des arguments est véridique. Si les deux arguments sont faux, évaluez le deuxième argument.

```
x = True
y = True
z = x or y # z = True
```

```
x = True
y = False
z = x or y # z = True

x = False
y = True
z = x or y # z = True

x = False
y = False
z = x or y # z = False

x = 1
y = 1
z = x or y # z = x, so z = 1, see `and` and `or` are not guaranteed to be a boolean

x = 1
y = 0
z = x or y # z = x, so z = 1 (see above)

x = 0
y = 1
z = x or y # z = y, so z = 1 (see above)

x = 0
y = 0
z = x or y # z = y, so z = 0 (see above)
```

Les `1` dans l'exemple ci-dessus peuvent être changés en n'importe quelle valeur, et les `0` peuvent être changés en n'importe quelle valeur.

ne pas

Il renvoie le contraire de l'instruction suivante:

```
x = True
y = not x # y = False

x = False
y = not x # y = True
```

Évaluation du court-circuit

Python évalue minimalement les expressions booléennes.

```
>>> def true_func():
...     print("true_func()")
...     return True
...
>>> def false_func():
...     print("false_func()")
...     return False
...
>>> true_func() or false_func()
true_func()
True
```

```
>>> false_func() or true_func()
false_func()
true_func()
True
>>> true_func() and false_func()
true_func()
false_func()
False
>>> false_func() and false_func()
false_func()
False
```

`et` et` ou` ne sont pas garantis pour renvoyer un booléen

Lorsque vous utilisez `or` , il renverra la première valeur de l'expression si elle est vraie, sinon elle retournera aveuglément la deuxième valeur. Ie `or` est équivalent à:

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

Pour `and` , il retournera sa première valeur si elle est fausse, sinon elle retourne la dernière valeur:

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

Un exemple simple

En Python, vous pouvez comparer un seul élément en utilisant deux opérateurs binaires - un de chaque côté:

```
if 3.14 < x < 3.142:
    print("x is near pi")
```

Dans de nombreux langages de programmation (la plupart?), Cela serait évalué d'une manière contraire aux calculs normaux: `(3.14 < x) < 3.142` , mais en Python, il est traité comme `3.14 < x and x < 3.142` , comme la plupart des non-programmeurs attendrait.

Lire Opérateurs booléens en ligne: https://riptutorial.com/fr/python/topic/1731/operateurs-booleens

Chapitre 139: Opérateurs mathématiques simples

Introduction

Python fait lui-même appel à des opérateurs mathématiques communs, y compris la division entière et flottante, la multiplication, l'exponentiation, l'addition et la soustraction. Le module mathématique (inclus dans toutes les versions standard de Python) offre des fonctionnalités étendues telles que les fonctions trigonométriques, les opérations root, les logarithmes, etc.

Remarques

Types numériques et leurs métaclasses

Le module de `numbers` contient les métaclasses abstraites pour les types numériques:

sous-classes	numéros.Nombre	nombres.intégrale	chiffres.Rational	numéros.Real	non
bool	✓	✓	✓	✓	✓
l'int	✓	✓	✓	✓	✓
fractions.Fraction	✓	-	✓	✓	✓
flotte	✓	-	-	✓	✓
complexe	✓	-	-	-	✓
décimal.Décimal	✓	-	-	-	-

Examples

Une addition

```
a, b = 1, 2

# Using the "+" operator:
a + b                   # = 3

# Using the "in-place" "+=" operator to add and assign:
a += b                  # a = 3 (equivalent to a = a + b)

import operator         # contains 2 argument arithmetic functions for the examples
```

```
operator.add(a, b)     # = 5  since a is set to 3 right before this line

# The "+=" operator is equivalent to:
a = operator.iadd(a, b)    # a = 5 since a is set to 3 right before this line
```

Combinaisons possibles (types intégrés):

- `int` et `int` (donne un `int`)
- `int` et `float` (donne un `float`)
- `int` et `complex` (donne un `complex`)
- `float` et `float` (donne un `float`)
- `float` et `complex` (donne un `complex`)
- `complex` et `complex` (donne un `complex`)

Remarque: l'opérateur `+` est également utilisé pour concaténer des chaînes, des listes et des tuples:

```
"first string " + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

Soustraction

```
a, b = 1, 2

# Using the "-" operator:
b - a                  # = 1


import operator        # contains 2 argument arithmetic functions
operator.sub(b, a)     # = 1
```

Combinaisons possibles (types intégrés):

- `int` et `int` (donne un `int`)
- `int` et `float` (donne un `float`)
- `int` et `complex` (donne un `complex`)
- `float` et `float` (donne un `float`)
- `float` et `complex` (donne un `complex`)
- `complex` et `complex` (donne un `complex`)

Multiplication

```
a, b = 2, 3

a * b                  # = 6

import operator
```

```
operator.mul(a, b)      # = 6
```

Combinaisons possibles (types intégrés):

- `int` et `int` (donne un `int`)
- `int` et `float` (donne un `float`)
- `int` et `complex` (donne un `complex`)
- `float` et `float` (donne un `float`)
- `float` et `complex` (donne un `complex`)
- `complex` et `complex` (donne un `complex`)

Remarque: L'opérateur `*` est également utilisé pour la concaténation répétée de chaînes, de listes et de n-uplets:

```
3 * 'ab'  # = 'abababab'
3 * ('a', 'b')  # = ('a', 'b', 'a', 'b', 'a', 'b')
```

Division

Python fait une division entière lorsque les deux opérandes sont des entiers. Le comportement des opérateurs de division de Python a changé depuis Python 2.x et 3.x (voir aussi Integer Division).

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x 2.7

En Python 2, le résultat de l'opérateur '/' dépend du type du numérateur et du dénominateur.

```
a / b                   # = 1

a / c                   # = 1.5

d / b                   # = -2

b / a                   # = 0

d / e                   # = -1
```

Notez que comme `a` et `b` sont `int` s, le résultat est un `int` .

Le résultat est toujours arrondi au sol.

Parce que `c` est un flottant, le résultat de `a / c` est un `float` .

Vous pouvez également utiliser le module opérateur:

```
import operator         # the operator module provides 2-argument arithmetic functions
operator.div(a, b)      # = 1
```

```
operator.__div__(a, b) # = 1
```

Python 2.x 2.2

Et si vous voulez la division float:

Conseillé:

```
from __future__ import division # applies Python 3 style division to the entire module
a / b                   # = 1.5
a // b                  # = 1
```

Ok (si vous ne voulez pas appliquer à tout le module):

```
a / (b * 1.0)           # = 1.5
1.0 * a / b             # = 1.5
a / b * 1.0             # = 1.0    (careful with order of operations)

from operator import truediv
truediv(a, b)           # = 1.5
```

Non recommandé (peut déclencher TypeError, par exemple si l'argument est complexe):

```
float(a) / b            # = 1.5
a / float(b)            # = 1.5
```

Python 2.x 2.2

L'opérateur '//' dans Python 2 force la division paralysée quel que soit le type.

```
a // b                # = 1
a // c                # = 1.0
```

Python 3.x 3.0

Dans Python 3, l'opérateur `/` effectue une division "true" indépendamment des types. L'opérateur `//` effectue la division d'étage et maintient le type.

```
a / b                   # = 1.5
e / b                   # = 5.0
a // b                  # = 1
a // c                  # = 1.0

import operator             # the operator module provides 2-argument arithmetic functions
operator.truediv(a, b)      # = 1.5
operator.floordiv(a, b)     # = 1
operator.floordiv(a, c)     # = 1.0
```

Combinaisons possibles (types intégrés):

- `int` et `int` (donne un `int` dans Python 2 et un `float` dans Python 3)
- `int` et `float` (donne un `float`)

- `int` et `complex` (donne un `complex`)
- `float` et `float` (donne un `float`)
- `float` et `complex` (donne un `complex`)
- `complex` et `complex` (donne un `complex`)

Voir PEP 238 pour plus d'informations.

Exponentation

```
a, b = 2, 3

(a ** b)                # = 8
pow(a, b)               # = 8

import math
math.pow(a, b)          # = 8.0 (always float; does not allow complex results)

import operator
operator.pow(a, b)      # = 8
```

Une autre différence entre le `pow` `math.pow` et `math.pow` est que le `pow` intégré peut accepter trois arguments:

```
a, b, c = 2, 3, 2

pow(2, 3, 2)            # 0, calculates (2 ** 3) % 2, but as per Python docs,
                        #    does so more efficiently
```

Fonctions spéciales

La fonction `math.sqrt(x)` calcule la racine carrée de `x` .

```
import math
import cmath
c = 4
math.sqrt(c)            # = 2.0 (always float; does not allow complex results)
cmath.sqrt(c)           # = (2+0j) (always complex)
```

Pour calculer d'autres racines, telles qu'une racine de cube, augmentez le nombre à l'inverse du degré de la racine. Cela pourrait être fait avec n'importe quelle fonction exponentielle ou opérateur.

```
 import math
 x = 8
 math.pow(x, 1/3) # evaluates to 2.0
 x**(1/3) # evaluates to 2.0
```

La fonction `math.exp(x)` calcule `e ** x` .

```
math.exp(0)  # 1.0
math.exp(1)  # 2.718281828459045 (e)
```

La fonction `math.expm1(x)` calcule `e ** x - 1` . Lorsque `x` est petit, cela donne une précision nettement meilleure que `math.exp(x) - 1` .

```
math.expm1(0)       # 0.0

math.exp(1e-6) - 1  # 1.0000004999621837e-06
math.expm1(1e-6)    # 1.0000005000001665e-06
# exact result      # 1.0000005000001666667083333341666...
```

Logarithmes

Par défaut, la fonction `math.log` calcule le logarithme d'un nombre, base e. Vous pouvez éventuellement spécifier une base comme second argument.

```
import math
import cmath

math.log(5)         # = 1.6094379124341003
# optional base argument. Default is math.e
math.log(5, math.e) # = 1.6094379124341003
cmath.log(5)        # = (1.6094379124341003+0j)
math.log(1000, 10)   # 3.0 (always returns float)
cmath.log(1000, 10)  # (3+0j)
```

Des variantes spéciales de la fonction `math.log` existent pour différentes bases.

```
# Logarithm base e - 1 (higher precision for low values)
math.log1p(5)       # = 1.791759469228055

# Logarithm base 2
math.log2(8)        # = 3.0

# Logarithm base 10
math.log10(100)     # = 2.0
cmath.log10(100)    # = (2+0j)
```

Opérations en place

Il est courant dans les applications d'avoir besoin d'un code comme celui-ci:

```
a = a + 1
```

ou

```
a = a * 2
```

Il existe un raccourci efficace pour ces opérations en place:

```
a += 1
# and
a *= 2
```

Tout opérateur mathématique peut être utilisé avant le caractère '=' pour effectuer une opération in-situ:

- `-=` décrémenter la variable en place
- `+=` incrémenter la variable en place
- `*=` multiplier la variable en place
- `/=` divise la variable en place
- `//=` sol divise la variable en place # Python 3
- `%=` renvoie le module de la variable en place
- `**=` élever à une puissance en place

D'autres opérateurs sur place existent pour les opérateurs de bits (`^` , `|` etc)

Fonctions trigonométriques

```
a, b = 1, 2

import math

math.sin(a)  # returns the sine of 'a' in radians
# Out: 0.8414709848078965

math.cosh(b)  # returns the inverse hyperbolic cosine of 'b' in radians
# Out: 3.7621956910836314

math.atan(math.pi)  # returns the arc tangent of 'pi' in radians
# Out: 1.2626272556789115

math.hypot(a, b) # returns the Euclidean norm, same as math.sqrt(a*a + b*b)
# Out: 2.23606797749979
```

> Notez que `math.hypot(x, y)` est également la longueur du vecteur (ou distance euclidienne) de l'origine `(0, 0)` au point `(x, y)` .
>
> Pour calculer la distance euclidienne entre deux points `(x1, y1)` & `(x2, y2)` vous pouvez utiliser `math.hypot` comme suit
>
> ```
> math.hypot(x2-x1, y2-y1)
> ```

Pour convertir des radians -> degrés et degrés -> les radians utilisent respectivement `math.degrees` et `math.radians`

```
math.degrees(a)
# Out: 57.29577951308232

math.radians(57.29577951308232)
# Out: 1.0
```

Module

Comme dans de nombreuses autres langues, Python utilise l'opérateur `%` pour calculer le module.

```
3 % 4      # 3
10 % 2     # 0
6 % 4      # 2
```

Ou en utilisant le module `operator` :

```
import operator

operator.mod(3 , 4)      # 3
operator.mod(10 , 2)     # 0
operator.mod(6 , 4)      # 2
```

Vous pouvez également utiliser des nombres négatifs.

```
-9 % 7      # 5
9 % -7      # -5
-9 % -7     # -2
```

Si vous avez besoin de trouver le résultat de la division et du module entiers, vous pouvez utiliser la fonction `divmod` comme raccourci:

```
quotient, remainder = divmod(9, 4)
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

Lire Opérateurs mathématiques simples en ligne:
https://riptutorial.com/fr/python/topic/298/operateurs-mathematiques-simples

Chapitre 140: Opérateurs sur les bits

Introduction

Les opérations binaires modifient les chaînes binaires au niveau des bits. Ces opérations sont incroyablement simples et sont directement prises en charge par le processeur. Ces quelques opérations sont nécessaires pour travailler avec des pilotes de périphériques, des graphiques bas niveau, la cryptographie et les communications réseau. Cette section fournit des connaissances et des exemples utiles des opérateurs de bits de Python.

Syntaxe

- x << y # Décalage bit à gauche
- x >> y # Changement binaire droit
- x & y # bit à bit ET
- x | y # bit à bit OU
- ~ x # bitwise non
- x ^ y # bit à bit

Examples

Bitwise AND

L'opérateur `&` exécutera un **AND** binaire, où un bit est copié s'il existe dans les **deux** opérandes. Cela signifie:

```
# 0 & 0 = 0
# 0 & 1 = 0
# 1 & 0 = 0
# 1 & 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 & 30
# Out: 28
# 28 = 0b11100

bin(60 & 30)
# Out: 0b11100
```

Bit à bit OU

Le `|` l'opérateur effectuera un "ou" binaire où un bit est copié s'il existe dans l'un des opérandes.

Cela signifie:

```
# 0 | 0 = 0
# 0 | 1 = 1
# 1 | 0 = 1
# 1 | 1 = 1

# 60 = 0b111100
# 30 = 0b011110
60 | 30
# Out: 62
# 62 = 0b111110

bin(60 | 30)
# Out: 0b111110
```

Bit à bit XOR (OU exclusif)

L'opérateur `^` exécutera un **XOR** binaire dans lequel un binaire `1` est copié si et seulement si c'est la valeur de exactement **un** opérande. Une autre manière de dire ceci est que le résultat est `1` seulement si les opérandes sont différents. Les exemples comprennent:

```
# 0 ^ 0 = 0
# 0 ^ 1 = 1
# 1 ^ 0 = 1
# 1 ^ 1 = 0

# 60 = 0b111100
# 30 = 0b011110
60 ^ 30
# Out: 34
# 34 = 0b100010

bin(60 ^ 30)
# Out: 0b100010
```

Décalage bit à gauche

L'opérateur `<<` effectuera un "décalage vers la gauche" au niveau du bit, la valeur de l'opérande gauche étant déplacée par le nombre de bits fournis par l'opérande droit.

```
# 2 = 0b10
2 << 2
# Out: 8
# 8 = 0b1000

bin(2 << 2)
# Out: 0b1000
```

L'exécution d'un décalage de `1` gauche équivaut à une multiplication par `2` :

```
7 << 1
# Out: 14
```

Effectuer un décalage de `n` de gauche équivaut à une multiplication par `2**n` :

```
3 << 4
# Out: 48
```

Changement bit à bit droit

L'opérateur `>>` effectuera un "décalage à droite" au niveau du bit, où la valeur de l'opérande gauche est déplacée vers la droite par le nombre de bits fournis par l'opérande droit.

```
# 8 = 0b1000
8 >> 2
# Out: 2
# 2 = 0b10

bin(8 >> 2)
# Out: 0b10
```

L'exécution d'un décalage de `1` droite équivaut à une division entière par `2` :

```
36 >> 1
# Out: 18

15 >> 1
# Out: 7
```

Effectuer un décalage de bit droit de `n` est équivalent à une division entière de `2**n` :

```
48 >> 4
# Out: 3

59 >> 3
# Out: 7
```

Pas au bit

L'opérateur `~` retournera tous les bits du numéro. Puisque les ordinateurs utilisent des représentations de nombres signés - plus particulièrement, la notation de complément à deux pour encoder des nombres binaires négatifs où les nombres négatifs sont écrits avec un premier (1) au lieu d'un zéro initial (0).

Cela signifie que si vous utilisiez 8 bits pour représenter vos deux nombres complémentaires, vous traiteriez les modèles de `0000 0000` à `0111 1111` pour représenter les nombres de 0 à 127 et réserver `1xxx xxxx` pour représenter les nombres négatifs.

Les nombres de complément à deux bits de huit bits

Morceaux	Valeur non signée	Valeur du complément à deux
0000 0000	0	0

Morceaux	Valeur non signée	Valeur du complément à deux
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
1111 1110	254	-2
1111 1111	255	-1

Essentiellement, cela signifie que `1010 0110` a une valeur non signée de 166 (obtenue en ajoutant `(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)`), il a une valeur de complément à deux de -90 (obtenue en ajoutant `(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)` , et en complément de la valeur).

De cette manière, les nombres négatifs sont réduits à -128 (`1000 0000`). Zéro (0) est représenté par `0000 0000` et moins un (-1) que `1111 1111` .

En général, cependant, cela signifie `~n = -n - 1` .

```
# 0 = 0b0000 0000
~0
# Out: -1
# -1 = 0b1111 1111

# 1 = 0b0000 0001
~1
# Out: -2
# -2 = 1111 1110

# 2 = 0b0000 0010
~2
# Out: -3
# -3 = 0b1111 1101

# 123 = 0b0111 1011
~123
# Out: -124
# -124 = 0b1000 0100
```

Notez que l'effet global de cette opération, appliqué aux nombres positifs, peut être résumé:

```
~n -> -|n+1|
```

Et puis, appliqué aux nombres négatifs, l'effet correspondant est:

```
~-n -> |n-1|
```

Les exemples suivants illustrent cette dernière règle ...

```
# -0 = 0b0000 0000
~-0
# Out: -1
# -1 = 0b1111 1111
# 0 is the obvious exception to this rule, as -0 == 0 always

# -1 = 0b1000 0001
~-1
# Out: 0
# 0 = 0b0000 0000

# -2 = 0b1111 1110
~-2
# Out: 1
# 1 = 0b0000 0001

# -123 = 0b1111 1011
~-123
# Out: 122
# 122 = 0b0111 1010
```

Opérations en place

Tous les opérateurs Bitwise (sauf `~`) ont leurs propres versions

```
a = 0b001
a &= 0b010
# a = 0b000

a = 0b001
a |= 0b010
# a = 0b011

a = 0b001
a <<= 2
# a = 0b100

a = 0b100
a >>= 2
# a = 0b001

a = 0b101
a ^= 0b011
# a = 0b110
```

Lire Opérateurs sur les bits en ligne: https://riptutorial.com/fr/python/topic/730/operateurs-sur-les-bits

Chapitre 141: Optimisation des performances

Remarques

Lorsque vous tentez d'améliorer les performances d'un script Python, vous devez avant tout trouver le goulot d'étranglement de votre script et noter qu'aucune optimisation ne peut compenser un mauvais choix dans les structures de données ou une faille dans la conception de votre algorithme. L'identification des goulots d'étranglement des performances peut être effectuée en profilant votre script. Deuxièmement, n'essayez pas d'optimiser trop tôt votre processus de codage au détriment de la lisibilité / conception / qualité. Donald Knuth a fait la déclaration suivante sur l'optimisation:

> «Nous devrions oublier les petites efficacités, disons environ 97% du temps: l'optimisation prématurée est la racine de tout mal. Pourtant, nous ne devrions pas laisser passer nos opportunités dans ce 3% critique.

Examples

Profilage de code

Tout d'abord, vous devriez être capable de trouver le goulot d'étranglement de votre script et noter qu'aucune optimisation ne peut compenser un mauvais choix dans la structure des données ou une faille dans la conception de votre algorithme. Deuxièmement, n'essayez pas d'optimiser trop tôt votre processus de codage au détriment de la lisibilité / conception / qualité. Donald Knuth a fait la déclaration suivante sur l'optimisation:

> "Nous devrions oublier les petites efficacités, disons environ 97% du temps: l'optimisation prématurée est la racine de tous les maux. Pourtant, nous ne devrions pas laisser passer nos opportunités dans ces 3% critiques"

Pour profiler votre code, vous disposez de plusieurs outils: `cProfile` (ou le `profile` plus lent) de la bibliothèque standard, `line_profiler` et `timeit` . Chacun d'eux sert un but différent.

`cProfile` est un profileur déterministe: l'appel de fonction, le retour de fonction et les événements d'exception sont surveillés, et des temporisations précises sont établies pour les intervalles entre ces événements (jusqu'à 0,001s). La documentation de la bibliothèque ([https://docs.python.org/2/library/profile.html][1]) nous fournit un cas d'utilisation simple

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

Ou si vous préférez envelopper des parties de votre code existant:

```
import cProfile, pstats, StringIO
```

```
pr = cProfile.Profile()
pr.enable()
# ... do something ...
# ... long ...
pr.disable()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

Cela créera des sorties ressemblant au tableau ci-dessous, où vous pourrez voir rapidement où votre programme passe le plus de temps et identifier les fonctions à optimiser.

```
         3 function calls in 0.000 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(f)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Le module `line_profiler` ([https://github.com/rkern/line_profiler]]) est utile pour avoir une analyse ligne par ligne de votre code. Ceci n'est évidemment pas gérable pour les scripts longs mais vise les extraits de code. Voir la documentation pour plus de détails. La manière la plus simple de commencer est d'utiliser le script kernprof comme expliqué sur la page du package, notez que vous devrez spécifier manuellement la ou les fonctions à profiler.

```
$ kernprof -l script_to_profile.py
```

kernprof va créer une instance de LineProfiler et l'insérer dans l'espace de noms `__builtins__` avec le profil name. Il a été écrit pour être utilisé comme décorateur, donc dans votre script, vous décorez les fonctions que vous souhaitez profiler avec `@profile` .

```
@profile
def slow_function(a, b, c):
    ...
```

Le comportement par défaut de kernprof consiste à placer les résultats dans un fichier binaire `script_to_profile.py.lprof` . Vous pouvez dire à kernprof d'afficher immédiatement les résultats formatés au terminal avec l'option [-v / - view]. Sinon, vous pouvez voir les résultats plus tard comme ceci:

```
$ python -m line_profiler script_to_profile.py.lprof
```

Finalement, `timeit` fournit un moyen simple de tester une ligne ou une petite expression à la fois à partir de la ligne de commande et du shell python. Ce module répondra à une question telle que, est-il plus rapide de faire une liste ou d'utiliser la `list()` intégrée `list()` lors de la transformation d'un ensemble en liste. Recherchez le mot-clé `setup` ou l'option `-s` pour ajouter le code d'installation.

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
```

d'un terminal

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
```

Lire Optimisation des performances en ligne:
https://riptutorial.com/fr/python/topic/5889/optimisation-des-performances

Chapitre 142: Oreiller

Examples

Lire le fichier image

```
from PIL import Image

im = Image.open("Image.bmp")
```

Convertir des fichiers en JPEG

```
from __future__ import print_function
import os, sys
from PIL import Image

for infile in sys.argv[1:]:
    f, e = os.path.splitext(infile)
    outfile = f + ".jpg"
    if infile != outfile:
        try:
            Image.open(infile).save(outfile)
        except IOError:
            print("cannot convert", infile)
```

Lire Oreiller en ligne: https://riptutorial.com/fr/python/topic/6841/oreiller

Chapitre 143: os.path

Introduction

Ce module implémente des fonctions utiles sur les chemins d'accès. Les paramètres de chemin peuvent être transmis sous forme de chaînes ou d'octets. Les applications sont encouragées à représenter les noms de fichiers en tant que chaînes de caractères (Unicode).

Syntaxe

- os.path.join (a, * p)
- os.path.basename (p)
- os.path.dirname (p)
- os.path.split (p)
- os.path.splitext (p)

Examples

Join Paths

Pour relier deux ou plusieurs composants de chemin, importez d'abord le module os de python, puis utilisez les éléments suivants:

```
import os
os.path.join('a', 'b', 'c')
```

L'avantage d'utiliser os.path est qu'il permet au code de rester compatible sur tous les systèmes d'exploitation, car il utilise le séparateur approprié pour la plate-forme sur laquelle il s'exécute.

Par exemple, le résultat de cette commande sous Windows sera:

```
>>> os.path.join('a', 'b', 'c')
'a\b\c'
```

Dans un OS Unix:

```
>>> os.path.join('a', 'b', 'c')
'a/b/c'
```

Chemin absolu du chemin relatif

Utilisez `os.path.abspath` :

```
>>> os.getcwd()
'/Users/csaftoiu/tmp'
```

```
>>> os.path.abspath('foo')
'/Users/csaftoiu/tmp/foo'
>>> os.path.abspath('../foo')
'/Users/csaftoiu/foo'
>>> os.path.abspath('/foo')
'/foo'
```

Manipulation de composants de chemin

Pour séparer un composant du chemin:

```
>>> p = os.path.join(os.getcwd(), 'foo.txt')
>>> p
'/Users/csaftoiu/tmp/foo.txt'
>>> os.path.dirname(p)
'/Users/csaftoiu/tmp'
>>> os.path.basename(p)
'foo.txt'
>>> os.path.split(os.getcwd())
('/Users/csaftoiu/tmp', 'foo.txt')
>>> os.path.splitext(os.path.basename(p))
('foo', '.txt')
```

Récupère le répertoire parent

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

Si le chemin donné existe.

vérifier si le chemin donné existe

```
path = '/home/john/temp'
os.path.exists(path)
#this returns false if path doesn't exist or if the path is a broken symbolic link
```

vérifier si le chemin donné est un répertoire, un fichier, un lien symbolique, un point de montage, etc.

vérifier si le chemin donné est un répertoire

```
dirname = '/home/john/python'
os.path.isdir(dirname)
```

vérifier si le chemin donné est un fichier

```
filename = dirname + 'main.py'
os.path.isfile(filename)
```

vérifier si le chemin donné est un lien symbolique

```
symlink = dirname + 'some_sym_link'
os.path.islink(symlink)
```

vérifier si le chemin donné est un point de montage

```
mount_path = '/home'
os.path.ismount(mount_path)
```

Lire os.path en ligne: https://riptutorial.com/fr/python/topic/1380/os-path

Chapitre 144: Outil 2to3

Syntaxe

- $ 2to3 [-options] chemin / vers / fichier.py

Paramètres

Paramètre	La description
filename / nom_répertoire	2to3 accepte une liste de fichiers ou de répertoires à transformer en argument. Les répertoires sont parcourus de manière récursive pour les sources Python.
Option	**Option Description**
-f FIX, --fix = FIX	Spécifier les transformations à appliquer par défaut: tous. Liste des transformations disponibles avec `--list-fixes`
-j PROCESSES, --processes = PROCESSES	Exécuter 2to3 simultanément
-x NOFIX, --nofix = NOFIX	Exclure une transformation
-l, --list-fixes	Liste des transformations disponibles
-p, --print-function	Changer la grammaire pour que `print()` soit considéré comme une fonction
-v, --verbose	Sortie plus détaillée
--no-diffs	Ne pas sortir les diffs du refactoring
-w	Ecrire les fichiers modifiés
-n, --nobackups	Ne créez pas de sauvegardes de fichiers modifiés
-o OUTPUT_DIR, --output-dir = OUTPUT_DIR	Placez les fichiers de sortie dans ce répertoire au lieu de remplacer les fichiers d'entrée. Nécessite l'indicateur `-n` , car les fichiers de sauvegarde ne sont pas nécessaires lorsque les fichiers d'entrée ne sont pas modifiés.
-W, --write-unchanged-files	Ecrire des fichiers de sortie même si aucun changement n'était requis. Utile avec `-o` pour qu'un arbre source complet soit traduit et copié. Implique `-w` .

Paramètre	La description
--add-suffix = ADD_SUFFIX	Spécifiez une chaîne à ajouter à tous les noms de fichiers de sortie. Requiert `-n` si non vide. Ex .: `--add-suffix='3'` va générer des fichiers `.py3` .

Remarques

L'outil 2to3 est un programme python utilisé pour convertir le code écrit en Python 2.x en code Python 3.x. L'outil lit le code source Python 2.x et applique une série de correcteurs pour le transformer en code Python 3.x valide.

L'outil 2to3 est disponible dans la bibliothèque standard sous le nom de lib2to3, qui contient un ensemble complet de correcteurs qui gèrent presque tout le code. Lib2to3 étant une bibliothèque générique, il est possible d'écrire vos propres correcteurs pour 2to3.

Examples

Utilisation de base

Considérez le code Python2.x suivant. Enregistrez le fichier sous le nom `example.py`

Python 2.x 2.0

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

Dans le fichier ci-dessus, il existe plusieurs lignes incompatibles. La méthode `raw_input()` a été remplacée par `input()` dans Python 3.x et `print` n'est plus une instruction, mais une fonction. Ce code peut être converti en code Python 3.x à l'aide de l'outil 2to3.

Unix

```
$ 2to3 example.py
```

les fenêtres

```
> path/to/2to3.py example.py
```

L'exécution du code ci-dessus affichera les différences par rapport au fichier source d'origine, comme indiqué ci-dessous.

```
RefactoringTool: Skipping implicit fixer: buffer
```

```
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored example.py
--- example.py    (original)
+++ example.py    (refactored)
@@ -1,5 +1,5 @@
 def greet(name):
-    print "Hello, {0}!".format(name)
-print "What's your name?"
-name = raw_input()
+    print("Hello, {0}!".format(name))
+print("What's your name?")
+name = input()
 greet(name)
RefactoringTool: Files that need to be modified:
RefactoringTool: example.py
```

Les modifications peuvent être réécrites dans le fichier source en utilisant le drapeau -w. Une sauvegarde du fichier d'origine appelé `example.py.bak` est créée, à moins que l'option -n ne soit indiquée.

Unix

```
$ 2to3 -w example.py
```

les fenêtres

```
> path/to/2to3.py -w example.py
```

Maintenant, le fichier `example.py` a été converti du code Python 2.x au code Python 3.x.

Une fois terminé, `example.py` contiendra le code Python3.x suivant:

Python 3.x 3.0

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

Lire Outil 2to3 en ligne: https://riptutorial.com/fr/python/topic/5320/outil-2to3

Chapitre 145: outil graphique

Introduction

Les outils python peuvent être utilisés pour générer des graphiques

Examples

PyDotPlus

PyDotPlus est une version améliorée de l'ancien projet pydot qui fournit une interface Python au langage Dot de Graphviz.

Installation

Pour la dernière version stable:

```
pip install pydotplus
```

Pour la version de développement:

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

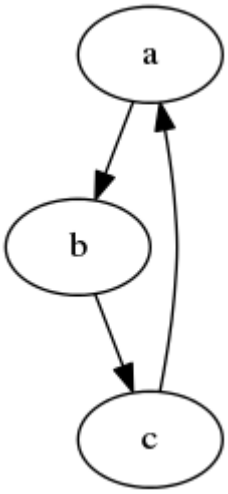
Charger le graphe tel que défini par un fichier DOT

- Le fichier est supposé être au format DOT. Il sera chargé, analysé et une classe Dot sera retournée, représentant le graphique. Par exemple, une simple démonstration.dot:

  digraph demo1 {a -> b -> c; c -> a; }

```
import pydotplus
graph_a = pydotplus.graph_from_dot_file('demo.dot')
graph_a.write_svg('test.svg') # generate graph in svg.
```

Vous obtiendrez un svg (Scalable Vector Graphics) comme ceci:



PyGraphviz

Récupérez PyGraphviz à partir de l'index du package Python à l'adresse http://pypi.python.org/pypi/pygraphviz

ou installez-le avec:

```
pip install pygraphviz
```

et une tentative sera faite pour trouver et installer une version appropriée qui correspond à votre système d'exploitation et à votre version de Python.

Vous pouvez installer la version de développement (sur github.com) avec:

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

Récupérez PyGraphviz à partir de l'index du package Python à l'adresse http://pypi.python.org/pypi/pygraphviz

ou installez-le avec:

```
easy_install pygraphviz
```

et une tentative sera faite pour trouver et installer une version appropriée qui correspond à votre système d'exploitation et à votre version de Python.

Charger le graphe tel que défini par un fichier DOT

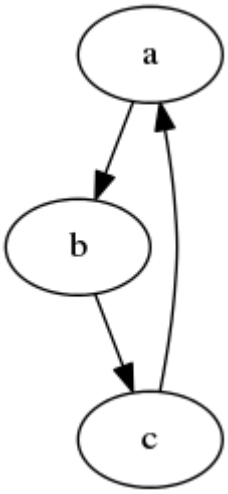
- Le fichier est supposé être au format DOT. Il sera chargé, analysé et une classe Dot sera retournée, représentant le graphique. Par exemple, une simple démonstration.dot:

  digraph demo1 {a -> b -> c; c -> a; }

- Chargez-le et dessinez-le.

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
G.draw('test', format='svg', prog='dot')
```

Vous obtiendrez un svg (Scalable Vector Graphics) comme ceci:



Lire outil graphique en ligne: https://riptutorial.com/fr/python/topic/9483/outil-graphique

Chapitre 146: Pandas Transform: préforme les opérations sur les groupes et concatène les résultats

Examples

Transformation simple

Tout d'abord, permet de créer un dataframe factice

Nous supposons qu'un client peut avoir n commandes, une commande peut avoir m articles et que les articles peuvent être commandés plusieurs fois

```
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1,1,1,1,1,2,2,3,3,3,3,3]
orders_df['order_id'] = [1,1,1,2,2,3,3,4,5,6,6,6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry',
'strawberry']

# And this is how the dataframe looks like:
print(orders_df)
#     customer_id  order_id        item
# 0             1         1      apples
# 1             1         1   chocolate
# 2             1         1   chocolate
# 3             1         2      coffee
# 4             1         2      coffee
# 5             2         3      apples
# 6             2         3     bananas
# 7             3         4      coffee
# 8             3         5   milkshake
# 9             3         6   chocolate
# 10            3         6  strawberry
# 11            3         6  strawberry
```

.
.

Nous allons maintenant utiliser la fonction de `transform` pandas pour compter le nombre de commandes par client

```
# First, we define the function that will be applied per customer_id
count_number_of_orders = lambda x: len(x.unique())

# And now, we can tranform each group using the logic defined above
orders_df['number_of_orders_per_cient'] = (              # Put the results into a new column
```

```
that is called 'number_of_orders_per_cient'
                    orders_df                              # Take the original dataframe
                   .groupby(['customer_id'])['order_id'] # Create a seperate group for each
customer_id & select the order_id
                   .transform(count_number_of_orders))    # Apply the function to each group
seperatly

# Inspecting the results ...
print(orders_df)
#     customer_id  order_id        item  number_of_orders_per_cient
# 0             1         1      apples                           2
# 1             1         1   chocolate                           2
# 2             1         1   chocolate                           2
# 3             1         2      coffee                           2
# 4             1         2      coffee                           2
# 5             2         3      apples                           1
# 6             2         3     bananas                           1
# 7             3         4      coffee                           3
# 8             3         5   milkshake                           3
# 9             3         6   chocolate                           3
# 10            3         6  strawberry                           3
# 11            3         6  strawberry                           3
```

Plusieurs résultats par groupe

Utilisation des fonctions de `transform` qui renvoient des sous-calculs par groupe

Dans l'exemple précédent, nous avions un résultat par client. Cependant, les fonctions renvoyant des valeurs différentes pour le groupe peuvent également être appliquées.

```
# Create a dummy dataframe
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1,1,1,1,1,2,2,3,3,3,3,3]
orders_df['order_id'] = [1,1,1,2,2,3,3,4,5,6,6,6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry',
'strawberry']


# Let's try to see if the items were ordered more than once in each orders

# First, we define a fuction that will be applied per group
def multiple_items_per_order(_items):
    # Apply .duplicated, which will return True is the item occurs more than once.
    multiple_item_bool = _items.duplicated(keep=False)
    return(multiple_item_bool)

# Then, we transform each group according to the defined function
orders_df['item_duplicated_per_order'] = (                    # Put the results into a new
column
                        orders_df                             # Take the orders dataframe
                        .groupby(['order_id'])['item']        # Create a seperate group for
each order_id & select the item
                        .transform(multiple_items_per_order)) # Apply the defined function to
```

```
each group separately

# Inspecting the results ...
print(orders_df)
#     customer_id  order_id        item  item_duplicated_per_order
# 0             1         1      apples                      False
# 1             1         1   chocolate                       True
# 2             1         1   chocolate                       True
# 3             1         2      coffee                       True
# 4             1         2      coffee                       True
# 5             2         3      apples                      False
# 6             2         3     bananas                      False
# 7             3         4      coffee                      False
# 8             3         5   milkshake                      False
# 9             3         6   chocolate                      False
# 10            3         6  strawberry                       True
# 11            3         6  strawberry                       True
```

Lire Pandas Transform: préforme les opérations sur les groupes et concatène les résultats en ligne: https://riptutorial.com/fr/python/topic/10947/pandas-transform--preforme-les-operations-sur-les-groupes-et-concatene-les-resultats

Chapitre 147: par groupe()

Introduction

Dans Python, la méthode `itertools.groupby()` permet aux développeurs de regrouper les valeurs d'une classe itérable en fonction d'une propriété spécifiée dans un autre ensemble de valeurs itérables.

Syntaxe

- itertools.groupby (itérable, clé = aucune ou une fonction)

Paramètres

Paramètre	Détails
itérable	Tout python itérable
clé	Fonction (critères) sur laquelle regrouper les itérables

Remarques

groupby () est délicat mais une règle générale à garder à l'esprit lors de son utilisation est la suivante:

Toujours trier les éléments que vous souhaitez regrouper avec la même clé que vous souhaitez utiliser pour le regroupement

Il est recommandé au lecteur de consulter la documentation ici et de voir comment cela est expliqué en utilisant une définition de classe.

Examples

Exemple 1

Dis que tu as la ficelle

```
s = 'AAAABBBCCDAABBB'
```

et vous voudriez le diviser pour que tous les A soient dans une liste et donc avec tous les B et C, etc. Vous pourriez faire quelque chose comme ça

```
s = 'AAAABBBCCDAABBB'
```

```
s_dict = {}
for i in s:
    if i not in s_dict.keys():
        s_dict[i] = [i]
    else:
        s_dict[i].append(i)
s_dict
```

Résulte en

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'],
 'B': ['B', 'B', 'B', 'B', 'B', 'B'],
 'C': ['C', 'C'],
 'D': ['D']}
```

Mais pour un grand ensemble de données, vous créeriez ces éléments en mémoire. C'est à ce moment que groupby () entre

Nous pourrions obtenir le même résultat de manière plus efficace en procédant comme suit

```
# note that we get a {key : value} pair for iterating over the items just like in python
dictionary
from itertools import groupby
s = 'AAAABBBCCDAABBB'
c = groupby(s)

dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Résulte en

```
{'A': ['A', 'A'], 'B': ['B', 'B', 'B'], 'C': ['C', 'C'], 'D': ['D']}
```

Notez que le nombre de «A» dans le résultat lorsque nous avons utilisé le groupe par est inférieur au nombre réel de «A» dans la chaîne d'origine. Nous pouvons éviter cette perte d'information en triant les éléments dans s avant de les transmettre à c comme indiqué ci-dessous.

```
c = groupby(sorted(s))

dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Résulte en

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'], 'B': ['B', 'B', 'B', 'B', 'B', 'B'], 'C': ['C', 'C'],
'D': ['D']}
```

Maintenant, nous avons tous nos A.

Exemple 2

Cet exemple illustre comment la clé par défaut est choisie si nous ne spécifions aucun

```
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('persons', 'man', 'woman')])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Résulte en

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Notez ici que le tuple dans son ensemble compte comme une clé dans cette liste

Exemple 3

Remarquez dans cet exemple que mulato et camel n'apparaissent pas dans notre résultat. Seul le dernier élément avec la clé spécifiée apparaît. Le dernier résultat pour c efface en fait deux résultats précédents. Mais regardez la nouvelle version où les données sont triées en premier sur la même clé.

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
c = groupby(list_things, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Résulte en

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Version triée

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
```

```
sorted_list = sorted(list_things, key = lambda x: x[0])
print(sorted_list)
print()
c = groupby(sorted_list, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Résulte en

```
['cow', 'cat', 'camel', 'dog', 'donkey', 'goat', 'mulato', 'mongoose', 'malloo', ('persons',
'man', 'woman'), 'wombat']

{'c': ['cow', 'cat', 'camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mulato', 'mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Exemple 4

Dans cet exemple, nous voyons ce qui se passe lorsque nous utilisons différents types d'itération.

```
things = [("animal", "bear"), ("animal", "duck"), ("plant", "cactus"), ("vehicle", "harley"),
\
          ("vehicle", "speed boat"), ("vehicle", "school bus")]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Résulte en

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
 'plant': [('plant', 'cactus')],
 'vehicle': [('vehicle', 'harley'),
  ('vehicle', 'speed boat'),
  ('vehicle', 'school bus')]}
```

Cet exemple ci-dessous est essentiellement le même que celui ci-dessus. La seule différence est que j'ai changé tous les tuples en listes.

```
things = [["animal", "bear"], ["animal", "duck"], ["vehicle", "harley"], ["plant", "cactus"],
\
          ["vehicle", "speed boat"], ["vehicle", "school bus"]]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Résultats

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
 'plant': [['plant', 'cactus']],
 'vehicle': [['vehicle', 'harley'],
  ['vehicle', 'speed boat'],
  ['vehicle', 'school bus']]}
```

Lire par groupe() en ligne: https://riptutorial.com/fr/python/topic/8690/par-groupe--

Chapitre 148: Persistance python

Syntaxe

- pickle.dump (obj, fichier, protocole = Aucun, *, fix_imports = True)
- pickle.load (fichier, *, fix_imports = True, encoding = "ASCII", errors = "strict")

Paramètres

Paramètre	Détails
obj	représentation décapée de obj dans le fichier objet de fichier ouvert
protocole	un entier, indique au pickler d'utiliser le protocole donné, `0` -ASCII, `1` - ancien format binaire
fichier	L'argument de fichier doit avoir une méthode write () `wb` pour la méthode de *vidage* et pour charger la méthode read () `rb`

Examples

Persistance python

Les objets tels que les nombres, les listes, les dictionnaires, les structures imbriquées et les objets d'instance de classe vivent dans la mémoire de votre ordinateur et sont perdus dès que le script se termine.

pickle stocke les données de manière persistante dans un fichier séparé.

la représentation décapée d'un objet est toujours un objet octets dans tous les cas, il faut donc ouvrir les fichiers dans `wb` pour stocker les données et `rb` pour charger les données à partir de pickle.

les données peuvent être différentes, par exemple,

```
data={'a':'some_value',
     'b':[9,4,7],
     'c':['some_str','another_str','spam','ham'],
     'd':{'key':'nested_dictionary'},
     }
```

Stocker des données

```
import pickle
file=open('filename','wb')  #file object in binary write mode
```

```
pickle.dump(data,file)      #dump the data in the file object
file.close()                #close the file to write into the file
```

Charger des données

```
import pickle
file=open('filename','rb')  #file object in binary read mode
data=pickle.load(file)      #load the data back
file.close()

>>>data
{'b': [9, 4, 7], 'a': 'some_value', 'd': {'key': 'nested_dictionary'},
 'c': ['some_str', 'another_str', 'spam', 'ham']}
```

Les types suivants peuvent être décapés

1. Aucun, vrai et faux
2. nombres entiers, nombres à virgule flottante, nombres complexes
3. chaînes, octets, bytearys
4. tuples, listes, ensembles et dictionnaires contenant uniquement des objets picklable
5. fonctions définies au niveau supérieur d'un module (en utilisant def, pas lambda)
6. fonctions intégrées définies au niveau supérieur d'un module
7. classes définies au niveau supérieur d'un module
8. instances de telles classes dont le **dict** ou le résultat de l'appel à **getstate** ()

Utilitaire de fonction pour enregistrer et charger

Enregistrer les données vers et depuis le fichier

```
import pickle
def save(filename,object):
    file=open(filename,'wb')
    pickle.dump(object,file)
    file.close()

def load(filename):
    file=open(filename,'rb')
    object=pickle.load(file)
    file.close()
    return object


>>>list_object=[1,1,2,3,5,8,'a','e','i','o','u']
>>>save(list_file,list_object)
>>>new_list=load(list_file)
>>>new_list
[1, 1, 2, 3, 5, 8, 'a', 'e', 'i', 'o', 'u'
```

Lire Persistance python en ligne: https://riptutorial.com/fr/python/topic/7810/persistance-python

Chapitre 149: Pièges courants

Introduction

Python est un langage destiné à être clair et lisible sans aucune ambiguïté ni comportement inattendu. Malheureusement, ces objectifs ne sont pas réalisables dans tous les cas, et c'est pourquoi Python a quelques cas isolés où il pourrait faire quelque chose de différent de ce que vous attendiez.

Cette section vous montrera quelques problèmes que vous pourriez rencontrer lors de l'écriture de code Python.

Examples

Changer la séquence sur laquelle vous parcourez

Une boucle `for` une itération sur une séquence, donc la **modification de cette séquence dans la boucle peut entraîner des résultats inattendus** (en particulier lors de l'ajout ou de la suppression d'éléments):

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]
```

Remarque: `list.pop()` est utilisé pour supprimer des éléments de la liste.

Le second élément n'a pas été supprimé car l'itération passe par les index dans l'ordre. La boucle ci-dessus est itérée deux fois, avec les résultats suivants:

```
# Iteration #1
index = 0
alist = [0, 1, 2]
alist.pop(0) # removes '0'

# Iteration #2
index = 1
alist = [1, 2]
alist.pop(1) # removes '2'

# loop terminates, but alist is not empty:
alist = [1]
```

Ce problème se pose parce que les indices changent tout en itérant dans la direction de l'indice croissant. Pour éviter ce problème, vous pouvez **parcourir la boucle en arrière** :

```
alist = [1,2,3,4,5,6,7]
```

```
for index, item in reversed(list(enumerate(alist))):
    # delete all even items
    if item % 2 == 0:
        alist.pop(index)
print(alist)
# Out: [1, 3, 5, 7]
```

En parcourant la boucle à partir de la fin, à mesure que les éléments sont supprimés (ou ajoutés), cela n'affecte pas les index des éléments figurant plus haut dans la liste. Donc, cet exemple va supprimer correctement tous les éléments qui sont même de `alist` .

Un problème similaire se pose lors de l' **insertion ou de l'ajout d'éléments à une liste sur laquelle vous parcourez** , ce qui peut entraîner une boucle infinie:

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    # break to avoid infinite loop:
    if index == 20:
        break
    alist.insert(index, 'a')
print(alist)
# Out (abbreviated): ['a', 'a', ..., 'a', 'a',  0,   1,   2]
```

Sans la condition de `break` , la boucle insérait `'a'` tant que l'ordinateur ne manquait pas de mémoire et que le programme est autorisé à continuer. Dans une situation comme celle-ci, il est généralement préférable de créer une nouvelle liste et d'ajouter des éléments à la nouvelle liste lorsque vous parcourez la liste d'origine.

Lors de l'utilisation d'une boucle `for` , **vous ne pouvez pas modifier les éléments de liste avec la variable d'espace réservé** :

```
alist = [1,2,3,4]
for item in alist:
    if item % 2 == 0:
        item = 'even'
print(alist)
# Out: [1,2,3,4]
```

Dans l'exemple ci-dessus, la **modification d'un `item` ne change rien à la liste d'origine** . Vous devez utiliser l'index de liste (`alist[2]`), et `enumerate()` fonctionne bien pour cela:

```
alist = [1,2,3,4]
for index, item in enumerate(alist):
    if item % 2 == 0:
        alist[index] = 'even'
print(alist)
# Out: [1, 'even', 3, 'even']
```

A **`while` boucle** pourrait être un meilleur choix dans certains cas:

Si vous souhaitez **supprimer tous les éléments** de la liste:

```
zlist = [0, 1, 2]
while zlist:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
#      2
# After: zlist = []
```

Bien que la simple réinitialisation de `zlist` produira le même résultat;

```
zlist = []
```

L'exemple ci-dessus peut également être combiné avec `len()` pour s'arrêter après un certain point ou pour supprimer tous les éléments sauf `x` de la liste:

```
zlist = [0, 1, 2]
x = 1
while len(zlist) > x:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)

# Out: 0
#      1
# After: zlist = [2]
```

Ou pour **parcourir une liste tout en supprimant des éléments répondant à une certaine condition** (dans ce cas, supprimer tous les éléments pairs):

```
zlist = [1,2,3,4,5]
i = 0
while i < len(zlist):
    if zlist[i] % 2 == 0:
        zlist.pop(i)
    else:
        i += 1
print(zlist)
# Out: [1, 3, 5]
```

Notez que vous n'incrémentez pas `i` après avoir supprimé un élément. En supprimant l'élément dans `zlist[i]` , l'index de l'élément suivant a diminué de un. En cochant `zlist[i]` avec la même valeur pour `i` lors de la prochaine itération, vous vérifierez correctement l'élément suivant de la liste. .

Une manière contraire de penser à supprimer des éléments indésirables d'une liste consiste à **ajouter les éléments souhaités à une nouvelle liste** . L'exemple suivant est une alternative à celui - ci `while` en exemple de la boucle:

```
zlist = [1,2,3,4,5]

z_temp = []
for item in zlist:
    if item % 2 != 0:
        z_temp.append(item)
zlist = z_temp
print(zlist)
# Out: [1, 3, 5]
```

Ici, nous canalisons les résultats souhaités dans une nouvelle liste. Nous pouvons alors éventuellement réaffecter la liste temporaire à la variable d'origine.

Avec cette tendance, vous pouvez invoquer l'une des fonctionnalités les plus élégantes et les plus puissantes de Python, la **compréhension de liste** , qui élimine les listes temporaires et diffère de l'idéologie des mutations de la liste / index déjà abordée.

```
zlist = [1,2,3,4,5]
[item for item in zlist if item % 2 != 0]
# Out: [1, 3, 5]
```

Argument par défaut mutable

```
def foo(li=[]):
    li.append(1)
    print(li)

foo([2])
# Out: [2, 1]
foo([3])
# Out: [3, 1]
```

Ce code se comporte comme prévu, mais si on ne passe pas un argument?

```
foo()
# Out: [1] As expected...

foo()
# Out: [1, 1]  Not as expected...
```

En effet, les arguments par défaut des fonctions et des méthodes sont évalués au moment de la **définition** plutôt que lors de l'exécution. Nous n'avons donc qu'une seule instance de la liste `li` .

La manière de le contourner est d'utiliser uniquement des types immuables pour les arguments par défaut:

```
def foo(li=None):
    if not li:
        li = []
    li.append(1)
    print(li)

foo()
```

```
# Out: [1]

foo()
# Out: [1]
```

Bien que ce soit une amélioration et même `if not li` correctement évaluée à `False` , de nombreux autres objets le font également, comme les séquences de longueur nulle. Les arguments suivants peuvent entraîner des résultats inattendus:

```
x = []
foo(li=x)
# Out: [1]

foo(li="")
# Out: [1]

foo(li=0)
# Out: [1]
```

L'approche idiomatique consiste à vérifier directement l'argument contre l'objet `None` :

```
def foo(li=None):
    if li is None:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]
```

Liste de multiplication et références communes

Prenons le cas de la création d'une structure de liste imbriquée en multipliant:

```
li = [[]] * 3
print(li)
# Out: [[], [], []]
```

À première vue, nous pensons avoir une liste de trois listes imbriquées différentes. Essayons d'ajouter `1` au premier:

```
li[0].append(1)
print(li)
# Out: [[1], [1], [1]]
```

`1` mais j'ai ajoutées à toutes les listes `li` .

La raison en est que `[[]] * 3` ne crée pas une `list` de 3 `list` différentes. Au contraire, il crée une `list` contenant 3 références au même objet `list` . En tant que tel, lorsque nous ajoutons à `li[0]` le changement est visible dans tous les sous-éléments de `li` . Ceci est équivalent à:

```
li = []
```

```
element = [[]]
li = element + element + element
print(li)
# Out: [[], [], []]
element.append(1)
print(li)
# Out: [[1], [1], [1]]
```

Cela peut être corroboré si nous imprimons les adresses mémoire de la `list` contenue en utilisant `id` :

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# Out: [6830760, 6830760, 6830760]
```

La solution consiste à créer les listes internes avec une boucle:

```
li = [[] for _ in range(3)]
```

Au lieu de créer une `list` unique et d'y faire 3 références, nous créons maintenant 3 listes distinctes. Ceci, encore une fois, peut être vérifié en utilisant la fonction `id` :

```
print([id(inner_list) for inner_list in li])
# Out: [6331048, 6331528, 6331488]
```

Vous pouvez aussi le faire. Il provoque une nouvelle liste vide à créer dans chaque `append` appel.

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> for k in li: print(id(k))
...
4315469256
4315564552
4315564808
```

N'utilisez pas d'index pour faire une boucle sur une séquence.

Ne pas:

```
for i in range(len(tab)):
    print(tab[i])
```

Faire :

```
for elem in tab:
    print(elem)
```

`for` va automatiser la plupart des opérations d'itération pour vous.

Utilisez énumérer si vous avez vraiment besoin de l'index et de l'élément .

```
for i, elem in enumerate(tab):
     print((i, elem))
```

Soyez prudent lorsque vous utilisez "==" pour vérifier si vrai ou faux

```
if (var == True):
    # this will execute if var is True or 1, 1.0, 1L

if (var != True):
    # this will execute if var is neither True nor 1

if (var == False):
    # this will execute if var is False or 0 (or 0.0, 0L, 0j)

if (var == None):
    # only execute if var is None

if var:
    # execute if var is a non-empty string/list/dictionary/tuple, non-0, etc

if not var:
    # execute if var is "", {}, [], (), 0, None, etc.

if var is True:
    # only execute if var is boolean True, not 1

if var is False:
    # only execute if var is boolean False, not 0

if var is None:
    # same as var == None
```

Ne vérifiez pas si vous le pouvez, faites-le simplement et gérez l'erreur

Les pythonistes disent généralement "C'est plus facile de demander le pardon que la permission".

Ne pas:

```
if os.path.isfile(file_path):
    file = open(file_path)
else:
    # do something
```

Faire:

```
try:
    file = open(file_path)
except OSError as e:
    # do something
```

Ou encore mieux avec `Python 2.6+` :

```
with open(file_path) as file:
```

C'est beaucoup mieux parce que c'est beaucoup plus générique. Vous pouvez appliquer `try/except` à presque tout. Vous n'avez pas besoin de vous soucier de ce qu'il faut faire pour le prévenir, prenez simplement soin de l'erreur que vous courez.

Ne pas vérifier avec le type

Python est typé dynamiquement, par conséquent, la vérification de type vous fait perdre de la flexibilité. Au lieu de cela, utilisez le typage de canard en vérifiant le comportement. Si vous attendez une chaîne dans une fonction, utilisez `str()` pour convertir n'importe quel objet en chaîne. Si vous vous attendez à une liste, utilisez `list()` pour convertir n'importe quelle liste pouvant être itérée.

Ne pas:

```
def foo(name):
    if isinstance(name, str):
        print(name.lower())

def bar(listing):
    if isinstance(listing, list):
        listing.extend((1, 2, 3))
        return ", ".join(listing)
```

Faire:

```
def foo(name) :
    print(str(name).lower())

def bar(listing) :
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

En utilisant la dernière manière, `foo` acceptera n'importe quel objet. `bar` acceptera les chaînes, les tuples, les ensembles, les listes et bien plus encore. Pas cher SEC.

Ne mélangez pas les espaces et les tabulations

Utiliser l'*objet* comme premier parent

C'est délicat, mais il va vous mordre à mesure que votre programme se développe. Il existe des classes anciennes et nouvelles dans `Python 2.x` Les anciens sont, eh bien, vieux. Ils manquent de certaines fonctionnalités et peuvent avoir un comportement difficile avec l'héritage. Pour être utilisable, n'importe laquelle de vos classes doit être du "nouveau style". Pour ce faire, faites-le hériter de l' `object` .

Ne pas:

```
class Father:
    pass

class Child(Father):
```

```
    pass
```

Faire:

```
class Father(object):
    pass


class Child(Father):
    pass
```

Dans `Python 3.x` toutes les classes sont de nouveau style, vous n'avez donc pas besoin de le faire.

Ne pas initialiser les attributs de classe en dehors de la méthode init

Les personnes venant d'autres langues le trouvent tentant car c'est ce que vous faites en Java ou en PHP. Vous écrivez le nom de la classe, puis listez vos attributs et donnez-leur une valeur par défaut. Il semble fonctionner en Python, cependant, cela ne fonctionne pas comme vous le pensez. Si vous configurez les attributs de classe (attributs statiques), vous obtiendrez sa valeur à moins que ce ne soit vide. Dans ce cas, il retournera les attributs de classe. Cela implique deux grands risques:

- Si l'attribut de classe est modifié, la valeur initiale est modifiée.
- Si vous définissez un objet mutable comme valeur par défaut, vous obtiendrez le même objet partagé entre les instances.

Ne (sauf si vous voulez statique):

```
class Car(object):
    color = "red"
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Faire :

```
class Car(object):
    def __init__(self):
        self.color = "red"
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

Entier et identité de chaîne

Python utilise la mise en cache interne pour une série d'entiers afin de réduire les surcharges inutiles résultant de leur création répétée.

En effet, cela peut conduire à un comportement déroutant lors de la comparaison d'identités entières:

```
>>> -8 is (-7 - 1)
```

```
False
>>> -3 is (-2 - 1)
True
```

et, en utilisant un autre exemple:

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

Attends quoi?

Nous pouvons voir que l'opération d'identité `is` le rendement `True` pour certains entiers (`-3` , `256`) , mais pas pour d' autres (`-8` , `257`).

Pour être plus précis, les entiers compris dans l'intervalle `[-5, 256]` sont mis en cache en interne lors du démarrage de l'interpréteur et ne sont créés qu'une seule fois. En tant que tels, ils sont **identiques** et la comparaison de leurs identités avec des rendements `is True` ; Les entiers en dehors de cette plage sont (généralement) créés à la volée et leurs identités se comparent à `False` .

C'est un piège courant, car il s'agit d'un test commun, mais assez souvent, le code échoue dans le processus intermédiaire ultérieur (ou pire - la production) sans raison apparente après avoir parfaitement fonctionné en développement.

La solution consiste à **toujours comparer les valeurs en utilisant l'** opérateur d' **égalité** (`==`) et **non** l'opérateur (`is`) d'identité.

Python conserve également des références à des chaînes couramment utilisées et peut entraîner un comportement confus similaire lors de la comparaison d'identités (c.-à-d. Utilisant `is`) de chaînes.

```
>>> 'python' is 'py' + 'thon'
True
```

La chaîne `'python'` est couramment utilisée, donc Python a un objet que toutes les références à la chaîne `'python'` utilisent.

Pour les chaînes peu communes, la comparaison d'identité échoue même lorsque les chaînes sont égales.

```
>>> 'this is not a common string' is 'this is not' + ' a common string'
False
>>> 'this is not a common string' == 'this is not' + ' a common string'
True
```

Ainsi, tout comme la règle pour les entiers, **comparez toujours les valeurs de chaîne en utilisant l'** opérateur d' **égalité** (`==`) et **non** l'opérateur d'identité (`is`).

Accéder aux attributs des littéraux int

Vous avez peut-être entendu dire que tout en Python est un objet, même littéral. Cela signifie, par exemple, que `7` est également un objet, ce qui signifie qu'il possède des attributs. Par exemple, l'un de ces attributs est la `bit_length` . Il renvoie la quantité de bits nécessaire pour représenter la valeur sur laquelle il est appelé.

```
x = 7
x.bit_length()
# Out: 3
```

En voyant le code ci-dessus fonctionner, vous pourriez penser intuitivement que `7.bit_length()` fonctionnerait aussi bien, seulement pour découvrir qu'il soulève une `SyntaxError` . Pourquoi? car l'interpréteur doit différencier un accès d'attribut et un nombre flottant (par exemple `7.2` ou `7.bit_length()`). Ce n'est pas possible, et c'est pourquoi une exception est soulevée.

Il y a plusieurs façons d'accéder aux attributs d'un littéral `int` :

```
# parenthesis
(7).bit_length()
# a space
7 .bit_length()
```

L'utilisation de deux points (comme ce `7..bit_length()`) ne fonctionne pas dans ce cas, car cela crée un littéral `float` et les flottants n'ont pas la méthode `bit_length()` .

Ce problème n’existe pas lorsqu’on accède aux attributs des littéraux `float` car l’interperter est suffisamment «intelligent» pour savoir qu’un littéral `float` ne peut pas en contenir deux `.` , par exemple:

```
7.2.as_integer_ratio()
# Out: (8106479329266893, 1125899906842624)
```

Chaînage ou opérateur

Lors du test de plusieurs comparaisons d'égalité:

```
if a == 3 or b == 3 or c == 3:
```

il est tentant d'abréger cela pour

```
if a or b or c == 3: # Wrong
```

C'est faux; le `or` a l' opérateur priorité inférieure à `==` , de sorte que l'expression est évaluée comme `if (a) or (b) or (c == 3):` . La manière correcte est de vérifier explicitement toutes les conditions:

```
if a == 3 or b == 3 or c == 3:  # Right Way
```

Alternativement, la fonction `any()` intégrée peut être utilisée à la place des chaînes `or` opérateurs:

```
if any([a == 3, b == 3, c == 3]): # Right
```

Ou, pour le rendre plus efficace:

```
if any(x == 3 for x in (a, b, c)): # Right
```

Ou, pour le raccourcir:

```
if 3 in (a, b, c): # Right
```

Ici, nous utilisons l'opérateur `in` pour tester si la valeur est présente dans un tuple contenant les valeurs que nous voulons comparer.

De même, il est incorrect d'écrire

```
if a == 1 or 2 or 3:
```

qui devrait être écrit comme

```
if a in (1, 2, 3):
```

sys.argv [0] est le nom du fichier en cours d'exécution

Le premier élément de `sys.argv[0]` est le nom du fichier python en cours d'exécution. Les éléments restants sont les arguments du script.

```
# script.py
import sys

print(sys.argv[0])
print(sys.argv)
```

```
$ python script.py
=> script.py
=> ['script.py']

$ python script.py fizz
=> script.py
=> ['script.py', 'fizz']

$ python script.py fizz buzz
=> script.py
=> ['script.py', 'fizz', 'buzz']
```

Les dictionnaires ne sont pas ordonnés

Vous pouvez vous attendre à ce qu'un dictionnaire Python soit trié par des clés comme, par exemple, un `std::map` C ++, mais ce n'est pas le cas:

```
myDict = {'first': 1, 'second': 2, 'third': 3}
print(myDict)
# Out: {'first': 1, 'second': 2, 'third': 3}

print([k for k in myDict])
# Out: ['second', 'third', 'first']
```

Python n'a pas de classe intégrée qui trie automatiquement ses éléments par clé.

Toutefois, si le tri n'est pas obligatoire et que vous souhaitez simplement que votre dictionnaire retienne l'ordre d'insertion de ses paires clé / valeur, vous pouvez utiliser des `collections.OrderedDict` :

```
from collections import OrderedDict

oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])

print([k for k in oDict])
# Out: ['first', 'second', 'third']
```

Gardez à l'esprit que l'initialisation d'un `OrderedDict` avec un dictionnaire standard ne vous permettra en aucun cas de trier le dictionnaire. Tout ce que fait cette structure est de *préserver* l'ordre d'insertion des clés.

L'implémentation des dictionnaires a été modifiée dans Python 3.6 pour améliorer leur consommation de mémoire. Un effet secondaire de cette nouvelle implémentation est qu'elle préserve également l'ordre des arguments de mots clés transmis à une fonction:

Python 3.x 3.6

```
def func(**kw): print(kw.keys())

func(a=1, b=2, c=3, d=4, e=5)
dict_keys(['a', 'b', 'c', 'd', 'e']) # expected order
```

> **Mise en** garde: méfiez-vous que *« l'aspect préservation de la commande de cette nouvelle implémentation est considéré comme un détail d'implémentation et il ne faut pas s'y fier »* , car cela pourrait changer à l'avenir.

Global Interpreter Lock (GIL) et les threads de blocage

Beaucoup de choses ont été écrites sur le GIL de Python . Cela peut parfois causer de la confusion lorsque vous traitez des applications multithread (à ne pas confondre avec les multiprocessus).

Voici un exemple:

```
import math
```

```
from threading import Thread

def calc_fact(num):
    math.factorial(num)

num = 600000
t = Thread(target=calc_fact, daemon=True, args=[num])
print("About to calculate: {}!".format(num))
t.start()
print("Calculating...")
t.join()
print("Calculated")
```

Vous vous attendriez à voir `Calculating...` imprimé immédiatement après le démarrage du thread, après tout, nous voulions que le calcul ait lieu dans un nouveau thread! Mais en réalité, vous le voyez imprimé une fois le calcul terminé. C'est parce que le nouveau thread s'appuie sur une fonction C (`math.factorial`) qui verrouille le GIL lors de son exécution.

Il y a plusieurs manières de contourner cela. La première consiste à implémenter votre fonction factorielle en Python natif. Cela permettra au thread principal de prendre le contrôle pendant que vous êtes dans votre boucle. L'inconvénient est que cette solution sera **beaucoup** plus lente, car nous n'utilisons plus la fonction C.

```
def calc_fact(num):
    """ A slow version of factorial in native Python """
    res = 1
    while num >= 1:
        res = res * num
        num -= 1
    return res
```

Vous pouvez également `sleep` pendant un certain temps avant de commencer votre exécution. Remarque: cela ne permettra pas à votre programme d'interrompre le calcul qui se produit à l'intérieur de la fonction C, mais cela permettra à votre thread principal de continuer après l'apparition, ce à quoi vous pouvez vous attendre.

```
def calc_fact(num):
    sleep(0.001)
    math.factorial(num)
```

Variable de fuite dans les listes compréhensibles et pour les boucles

Considérer la compréhension de liste suivante

Python 2.x 2.7

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 2
```

Cela se produit uniquement dans Python 2 en raison du fait que la compréhension de la liste «fuit» la variable de contrôle de la boucle dans la portée environnante (source). Ce

comportement peut conduire à des bogues difficiles à trouver et ***a été corrigé dans Python 3*** .

Python 3.x 3.0

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 0
```

De même, pour les boucles n'ont pas de portée privée pour leur variable d'itération

```
i = 0
for i in range(3):
    pass
print(i) # Outputs 2
```

Ce type de comportement se produit à la fois dans Python 2 et Python 3.

Pour éviter les problèmes avec les variables qui fuient, utilisez les nouvelles variables dans les listes compréhensibles et les boucles, le cas échéant.

Retour multiple

La fonction xyz renvoie deux valeurs a et b:

```
def xyz():
  return a, b
```

Le code appelant xyz stocke les résultats dans une variable en supposant que xyz ne renvoie qu'une seule valeur:

```
t = xyz()
```

La valeur de `t` est en fait un tuple (a, b), donc toute action sur `t` supposant que ce n'est pas un tuple peut échouer **profondément** dans le code avec une **erreur** inattendue sur les tuples.

> TypeError: le type tuple ne définit pas la méthode ...

La solution serait de faire:

```
a, b = xyz()
```

Les débutants auront du mal à trouver la raison de ce message en lisant uniquement le message d'erreur tuple!

Clés Pythonic JSON

```
my_var = 'bla';
api_key = 'key';
...lots of code here...
params = {"language": "en", my_var: api_key}
```

Si vous avez l'habitude de JavaScript, l'évaluation des variables dans les dictionnaires Python ne correspondra pas à vos attentes. Cette instruction en JavaScript entraînerait l'objet `params` comme suit:

```
{
    "language": "en",
    "my_var": "key"
}
```

En Python, cependant, le dictionnaire suivant apparaîtrait:

```
{
    "language": "en",
    "bla": "key"
}
```

`my_var` est évalué et sa valeur est utilisée comme clé.

Lire Pièges courants en ligne: https://riptutorial.com/fr/python/topic/3553/pieges-courants

Chapitre 150: pip: PyPI Package Manager

Introduction

pip est le gestionnaire de paquets le plus utilisé pour Python Package Index, installé par défaut avec les versions récentes de Python.

Syntaxe

- pip <command> [options] où <command> est l'un des suivants:
  - installer
    - Installer des paquets
  - désinstaller
    - Désinstaller des packages
  - gel
    - Sortie des packages installés dans le format des exigences
  - liste
    - Liste des paquets installés
  - montrer
    - Afficher des informations sur les packages installés
  - chercher
    - Rechercher des forfaits PyPI
  - roue
    - Construire des roues à partir de vos exigences
  - Zip *: français
    - Zip des paquets individuels (obsolète)
  - décompresser
    - Décompressez les paquets individuels (obsolètes)
  - paquet
    - Créer des pybundles (obsolètes)
  - Aidez-moi
    - Afficher l'aide pour les commandes

Remarques

Parfois, `pip` réalisera une compilation manuelle du code natif. Sous Linux, python choisira automatiquement un compilateur C disponible sur votre système. Reportez-vous au tableau ci-dessous pour obtenir la version requise de Visual Studio / Visual C ++ sous Windows (les nouvelles versions ne fonctionneront pas).

Version Python	Version de Visual Studio	Version de Visual C ++
2.6 - 3.2	Visual Studio 2008	Visual C ++ 9.0

Version Python	Version de Visual Studio	Version de Visual C ++
3.3 - 3.4	Visual Studio 2010	Visual C ++ 10.0
3.5	Visual Studio 2015	Visual C ++ 14.0

Source: wiki.python.org

Examples

Installer des paquets

Pour installer la dernière version d'un package nommé `SomePackage` :

```
$ pip install SomePackage
```

Pour installer une version spécifique d'un package:

```
$ pip install SomePackage==1.0.4
```

Pour spécifier une version minimale à installer pour un package:

```
$ pip install SomePackage>=1.0.4
```

Si les commandes montrent une autorisation refusée, une erreur sur Linux / Unix, puis utilisez `sudo` avec les commandes

---

Installer à partir des fichiers d'exigences

```
$ pip install -r requirements.txt
```

Chaque ligne du fichier d'exigences indique quelque chose à installer, et comme les arguments à installer dans Pip, les détails sur le format des fichiers sont ici: Format du fichier d'exigences .

Après avoir installé le paquet, vous pouvez le vérifier en utilisant la commande `freeze` :

```
$ pip freeze
```

Désinstaller des packages

Pour désinstaller un package:

```
$ pip uninstall SomePackage
```

Pour lister tous les paquets installés en utilisant `pip`

Pour répertorier les packages installés:

```
$ pip list
# example output
docutils (0.9.1)
Jinja2 (2.6)
Pygments (1.5)
Sphinx (1.1.2)
```

Pour répertorier les packages obsolètes et afficher la dernière version disponible:

```
$ pip list --outdated
# example output
docutils (Current: 0.9.1 Latest: 0.10)
Sphinx (Current: 1.1.2 Latest: 1.1.3)
```

Forfaits de mise à niveau

Fonctionnement

```
$ pip install --upgrade SomePackage
```

mettra à jour le paquet `SomePackage` et toutes ses dépendances. De plus, pip supprime automatiquement les anciennes versions du package avant la mise à niveau.

Pour mettre à jour pip lui-même, faites

```
$ pip install --upgrade pip
```

sur Unix ou

```
$ python -m pip install --upgrade pip
```

sur les machines Windows.

Mettre à jour tous les paquets obsolètes sous Linux

`pip` ne contient actuellement aucun indicateur permettant à un utilisateur de mettre à jour tous les paquets obsolètes en une seule fois. Cependant, ceci peut être accompli en combinant des commandes dans un environnement Linux:

```
pip list --outdated --local | grep -v '^\-e' | cut -d = -f 1  | xargs -n1 pip install -U
```

Cette commande prend tous les paquets de la virtual virtual locale et vérifie s'ils sont obsolètes. A partir de cette liste, il récupère le nom du package puis le transfère à une commande `pip install -U` . À la fin de ce processus, tous les packages locaux doivent être mis à jour.

Mettre à jour tous les paquets obsolètes sous Windows

`pip` ne contient actuellement aucun indicateur permettant à un utilisateur de mettre à jour tous les paquets obsolètes en une seule fois. Cependant, ceci peut être accompli en regroupant les commandes dans un environnement Windows:

```
for /F "delims= " %i in ('pip list --outdated --local') do pip install -U %i
```

Cette commande prend tous les paquets de la virtual virtual locale et vérifie s'ils sont obsolètes. A partir de cette liste, il récupère le nom du package puis le transfère à une commande `pip install -U` . À la fin de ce processus, tous les packages locaux doivent être mis à jour.

Créez un fichier requirements.txt de tous les packages du système

`pip` aide à créer des fichiers `requirements.txt` en fournissant l'option de `freeze` .

```
pip freeze > requirements.txt
```

Cela enregistrera une liste de tous les packages et de leur version installée sur le système dans un fichier nommé `requirements.txt` dans le dossier en cours.

Créer un fichier requirements.txt de packages uniquement dans la virtualenv actuelle

`pip` aide à créer des fichiers `requirements.txt` en fournissant l'option de `freeze` .

```
pip freeze --local > requirements.txt
```

Le paramètre `--local` affichera uniquement une liste de packages et de versions installés localement sur virtualenv. Les paquets globaux ne seront pas listés.

Utiliser une certaine version de Python avec pip

Si Python 3 et Python 2 sont tous deux installés, vous pouvez spécifier la version de Python que vous souhaitez que pip utilise. Ceci est utile lorsque les packages ne prennent en charge que Python 2 ou 3 ou lorsque vous souhaitez tester avec les deux.

Si vous souhaitez installer des packages pour Python 2, exécutez soit:

```
pip install [package]
```

ou:

```
pip2 install [package]
```

Si vous souhaitez installer des packages pour Python 3, procédez comme suit:

```
pip3 install [package]
```

Vous pouvez également appeler l'installation d'un package sur une installation python spécifique avec:

```
\path\to\that\python.exe -m pip install some_package # on Windows OR
/usr/bin/python25 -m pip install some_package # on OS-X/Linux
```

Sur les plates-formes OS-X / Linux / Unix, il est important de connaître la distinction entre la version système de python (dont la mise à niveau rend le système inutilisable) et la ou les versions utilisateur de python. Vous **pouvez** , *selon votre tentative de mise à niveau* , préfixer ces commandes avec `sudo` et entrer un mot de passe.

De même, sous Windows, certaines installations de python, en particulier celles qui font partie d'un autre package, peuvent être installées dans des répertoires système - ceux que vous devrez mettre à niveau à partir d'une fenêtre de commande fonctionnant en mode Admin - si Faites ceci c'est une **très** bonne idée de vérifier quelle installation python vous essayez de mettre à jour avec une commande telle que `python -c"import sys;print(sys.path);"` ou `py -3.5 -c"import sys;print(sys.path);"` vous pouvez également vérifier quel pip que vous essayez d'exécuter avec `pip --version`

Sous Windows, si python 2 et python 3 sont tous deux installés sur votre chemin et que votre python 3 est supérieur à 3,4, vous aurez probablement également le lanceur python `py` sur votre chemin système. Vous pouvez alors faire des trucs comme:

```
py -3 -m pip install -U some_package # Install/Upgrade some_package to the latest python 3
py -3.3 -m pip install -U some_package # Install/Upgrade some_package to python 3.3 if present
py -2 -m pip install -U some_package # Install/Upgrade some_package to the latest python 2 -
64 bit if present
py -2.7-32 -m pip install -U some_package # Install/Upgrade some_package to python 2.7 - 32
bit if present
```

Si vous exécutez et maintenez plusieurs versions de python, je vous recommande fortement de lire les environnements virtuels python `virtualenv` ou `venv` qui vous permettent d'isoler la version de python et les paquetages présents.

Installation de paquets pas encore sur pip sous forme de roues

De nombreux packages python pur ne sont pas encore disponibles sur Python Package Index en tant que roues, mais s'installent toujours correctement. Cependant, certains paquets sous Windows donnent l'erreur redoutée de vcvarsall.bat.

Le problème est que le package que vous tentez d'installer contient une extension C ou C ++ et qu'il n'est pas *actuellement* disponible en tant que roue pré-construite à partir de l'index python, *pypi* et que vous n'avez pas besoin de la chaîne de tels articles.

La solution la plus simple consiste à consulter l' excellent site de Christoph Gohlke et à trouver la version **appropriée** des bibliothèques dont vous avez besoin. En utilisant le nom de paquet **-cp *NN* -** doit correspondre à votre version de python, c'est-à-dire que si vous utilisez Windows 32 bits

python *même sur win64,* le nom doit inclure **-win32-** et si vous utilisez le python 64 bits, **-win_amd64 -** et la version python doit correspondre, par exemple pour Python 34 le nom du fichier **doit** inclure **-cp *34-,*** etc. est essentiellement la magie que pip fait pour vous sur le site pypi.

Alternativement, vous devez obtenir le kit de développement Windows correspondant à la version de python que vous utilisez, les en-têtes de toutes les bibliothèques avec lesquelles le package que vous essayez de construire interagissent, éventuellement les en-têtes python de la version de python, etc.

Python 2.7 utilisait Visual Studio 2008, Python 3.3 et 3.4 utilisaient Visual Studio 2010 et Python 3.5+ utilisait Visual Studio 2015.

- Installez « Package de compilateur Visual C ++ pour Python 2.7 », disponible sur le site Web de Microsoft **ou**
- Installez « Windows SDK for Windows 7 et .NET Framework 4 » (v7.1), disponible sur le site Web de Microsoft **ou**
- Installez Visual Studio 2015 Community Edition *(ou toute version ultérieure, lorsque celles-ci sont publiées)* , en **vous assurant de sélectionner les options d'installation du support C & C ++** *par défaut* . Le téléchargement et l'installation *peuvent durer jusqu'à* ***8 heures*** . alors assurez- **vous** que ces options sont définies au premier essai.

Ensuite, vous *devrez peut-être* localiser les fichiers d'en-tête, *lors de la révision correspondante* pour les bibliothèques auxquelles le package souhaité se connecte, et les télécharger vers les emplacements appropriés.

Enfin, vous pouvez laisser pip faire votre compilation - bien sûr, si le paquet a des dépendances que vous n'avez pas encore, vous devrez peut-être aussi trouver les fichiers d'en-tête pour eux aussi.

Alternatives: Cela vaut la peine de regarder, *aussi bien sur le site de pypi que sur celui de Christop* , pour toute version légèrement antérieure du paquet que vous cherchez qui soit purement python ou pré-construit pour votre plate-forme et version python. trouvé, jusqu'à ce que votre paquet devienne disponible. De même, si vous utilisez la toute dernière version de python, vous trouverez peut-être que les responsables des paquets ont **besoin d'** un peu de temps pour les rattraper. Pour les projets qui **nécessitent** un package spécifique, vous devrez peut-être utiliser un python légèrement plus ancien. Vous pouvez également vérifier le site source des packages pour voir s'il existe une version fourchue disponible pré-construite ou en python pur et rechercher des packages alternatifs offrant les fonctionnalités dont vous avez besoin mais qui sont disponibles. Oreiller , *activement maintenu* , en remplacement de PIL *actuellement non mis à jour en 6 ans et non disponible pour python 3* .

Après - **propos** , j'encourage tous ceux qui rencontrent ce problème à consulter le gestionnaire de bogues du paquet et à ajouter, ou augmenter s'il n'y en a pas déjà, un ticket demandant **poliment** aux responsables des paquets de fournir une roue sur pypi pour vos besoins spécifiques. combinaison de plate-forme et de python, si cela est fait alors normalement les choses iront mieux avec le temps, certains responsables de paquets ne réalisent pas qu'ils ont manqué une combinaison donnée que les gens pourraient utiliser.

Remarque sur l'installation de pré-versions

Pip suit les règles du contrôle de version sémantique et préfère par défaut les paquets publiés sur les pré-versions. Donc, si un paquet donné a été publié en tant que `V0.98` et qu'il y a aussi une version candidate `V1.0-rc1` le comportement par défaut de `pip install` sera d'installer `V0.98` - si vous souhaitez installer la version candidate, il *vous est conseillé Pour tester d'abord dans un environnement virtuel* , vous pouvez activer le faire avec `--pip install --pre` *-package-name* ou `--pip install --pre --upgrade` *nom-package* . Dans de nombreux cas, il est possible que les versions préliminaires ou les versions candidates ne disposent pas de roulettes conçues pour toutes les combinaisons de plates-formes et de versions, de sorte que vous êtes plus susceptible de rencontrer les problèmes ci-dessus.

Remarque sur l'installation des versions de développement

Vous pouvez également utiliser pip pour installer des versions de développement de paquets à partir de github et d'autres emplacements, car ce code est en flux, il est très peu probable que des roues soient construites pour cela, donc tous les paquets impurs nécessiteront la présence des outils de construction. être cassé à tout moment, l'utilisateur est **fortement** encouragé à installer uniquement de tels paquets dans un environnement virtuel.

Trois options existent pour de telles installations:

1. Télécharger un instantané compressé, la plupart des systèmes de contrôle de version en ligne ont la possibilité de télécharger un instantané compressé du code. Cela peut être téléchargé manuellement et ensuite installé avec le *chemin d'* `pip install` */ vers / téléchargé / fichier* note que pour la plupart des formats de compression, pip gérera le déballage dans une zone de cache, etc.
2. Laissez pip s'occuper du téléchargement avec: `pip install` *URL / of / package / repository* - vous devrez peut-être utiliser les `--trusted-host` , `--client-cert` et / ou `--proxy` pour que cela fonctionne. correctement, en particulier dans un environnement d'entreprise. par exemple:

```
    > py -3.5-32 -m venv demo-pip
    > demo-pip\Scripts\activate.bat
    > python -m pip install -U pip
    Collecting pip
      Using cached pip-9.0.1-py2.py3-none-any.whl
    Installing collected packages: pip
      Found existing installation: pip 8.1.1
        Uninstalling pip-8.1.1:
          Successfully uninstalled pip-8.1.1
    Successfully installed pip-9.0.1
    > pip install git+https://github.com/sphinx-doc/sphinx/
    Collecting git+https://github.com/sphinx-doc/sphinx/
      Cloning https://github.com/sphinx-doc/sphinx/ to c:\users\steve-
~1\appdata\local\temp\pip-04yn9hpp-build
    Collecting six>=1.5 (from Sphinx==1.7.dev20170506)
      Using cached six-1.10.0-py2.py3-none-any.whl
    Collecting Jinja2>=2.3 (from Sphinx==1.7.dev20170506)
      Using cached Jinja2-2.9.6-py2.py3-none-any.whl
    Collecting Pygments>=2.0 (from Sphinx==1.7.dev20170506)
```

```
      Using cached Pygments-2.2.0-py2.py3-none-any.whl
    Collecting docutils>=0.11 (from Sphinx==1.7.dev20170506)
      Using cached docutils-0.13.1-py3-none-any.whl
    Collecting snowballstemmer>=1.1 (from Sphinx==1.7.dev20170506)
      Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
    Collecting babel!=2.0,>=1.3 (from Sphinx==1.7.dev20170506)
      Using cached Babel-2.4.0-py2.py3-none-any.whl
    Collecting alabaster<0.8,>=0.7 (from Sphinx==1.7.dev20170506)
      Using cached alabaster-0.7.10-py2.py3-none-any.whl
    Collecting imagesize (from Sphinx==1.7.dev20170506)
      Using cached imagesize-0.7.1-py2.py3-none-any.whl
    Collecting requests>=2.0.0 (from Sphinx==1.7.dev20170506)
      Using cached requests-2.13.0-py2.py3-none-any.whl
    Collecting typing (from Sphinx==1.7.dev20170506)
      Using cached typing-3.6.1.tar.gz
    Requirement already satisfied: setuptools in f:\toolbuild\temp\demo-pip\lib\site-packages
(from Sphinx==1.7.dev20170506)
    Collecting sphinxcontrib-websupport (from Sphinx==1.7.dev20170506)
      Downloading sphinxcontrib_websupport-1.0.0-py2.py3-none-any.whl
    Collecting colorama>=0.3.5 (from Sphinx==1.7.dev20170506)
      Using cached colorama-0.3.9-py2.py3-none-any.whl
    Collecting MarkupSafe>=0.23 (from Jinja2>=2.3->Sphinx==1.7.dev20170506)
      Using cached MarkupSafe-1.0.tar.gz
    Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.7.dev20170506)
      Using cached pytz-2017.2-py2.py3-none-any.whl
    Collecting sqlalchemy>=0.9 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
      Downloading SQLAlchemy-1.1.9.tar.gz (5.2MB)
        100% |################################| 5.2MB 220kB/s
    Collecting whoosh>=2.0 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
      Downloading Whoosh-2.7.4-py2.py3-none-any.whl (468kB)
        100% |################################| 471kB 1.1MB/s
    Installing collected packages: six, MarkupSafe, Jinja2, Pygments, docutils,
snowballstemmer, pytz, babel, alabaster, imagesize, requests, typing, sqlalchemy, whoosh,
sphinxcontrib-websupport, colorama, Sphinx
      Running setup.py install for MarkupSafe ... done
      Running setup.py install for typing ... done
      Running setup.py install for sqlalchemy ... done
      Running setup.py install for Sphinx ... done
    Successfully installed Jinja2-2.9.6 MarkupSafe-1.0 Pygments-2.2.0 Sphinx-1.7.dev20170506
alabaster-0.7.10 babel-2.4.0 colorama-0.3.9 docutils-0.13.1 imagesize-0.7.1 pytz-2017.2
requests-2.13.0 six-1.10.0 snowballstemmer-1.2.1 sphinxcontrib-websupport-1.0.0 sqlalchemy-
1.1.9 typing-3.6.1 whoosh-2.7.4
```

Notez le préfixe `git+` sur l'URL.

3. Cloner le dépôt en utilisant `git` , `mercurial` ou tout autre outil acceptable, de *préférence un outil DVCS,* et utiliser l' `pip install` *path / to / clonée / repo* - ce sera à la **fois** processus tout fichier requires.text et effectuer la construction et étapes configuration, *vous pouvez modifier manuellement répertoire à votre référentiel cloné et exécutez* `pip install -r requires.txt` , *puis* `python setup.py install` *pour obtenir le même effet.* Les grands avantages de cette approche est que si l'opération de clonage initial peut prendre plus de temps que l'instantané téléchargement , vous pouvez mettre à jour la dernière avec, dans le cas de git: `git pull origin master` et si la version actuelle contient des erreurs , vous pouvez utiliser `pip uninstall` *nom-package,* puis utilisez les `git checkout` pour revenir à l'historique du référentiel vers les versions antérieures et réessayer.

Lire pip: PyPI Package Manager en ligne: https://riptutorial.com/fr/python/topic/1781/pip--pypi-

package-manager

Chapitre 151: Polymorphisme

Examples

Polymorphisme de base

Le polymorphisme est la capacité à exécuter une action sur un objet quel que soit son type. Ceci est généralement implémenté en créant une classe de base et en ayant deux ou plusieurs sous-classes qui implémentent toutes des méthodes avec la même signature. Toute autre fonction ou méthode manipulant ces objets peut appeler les mêmes méthodes, quel que soit le type d'objet sur lequel elle opère, sans avoir à effectuer au préalable une vérification de type. Dans la terminologie orientée objet, lorsque la classe X étend la classe Y, alors Y est appelée classe super ou classe de base et X est appelée sous-classe ou classe dérivée.

```
class Shape:
    """
    This is a parent class that is intended to be inherited by other classes
    """

    def calculate_area(self):
        """
        This method is intended to be overridden in subclasses.
        If a subclass doesn't implement it but it is called, NotImplemented will be raised.

        """
        raise NotImplemented

class Square(Shape):
    """
    This is a subclass of the Shape class, and represents a square
    """
    side_length = 2     # in this example, the sides are 2 units long

    def calculate_area(self):
        """
        This method overrides Shape.calculate_area(). When an object of type
        Square has its calculate_area() method called, this is the method that
        will be called, rather than the parent class' version.

        It performs the calculation necessary for this shape, a square, and
        returns the result.
        """
        return self.side_length * 2

class Triangle(Shape):
    """
    This is also a subclass of the Shape class, and it represents a triangle
    """
    base_length = 4
    height = 3

    def calculate_area(self):
        """
        This method also overrides Shape.calculate_area() and performs the area
```

```
        calculation for a triangle, returning the result.
        """

        return 0.5 * self.base_length * self.height

def get_area(input_obj):
    """
    This function accepts an input object, and will call that object's
    calculate_area() method. Note that the object type is not specified. It
    could be a Square, Triangle, or Shape object.
    """

    print(input_obj.calculate_area())

# Create one object of each class
shape_obj = Shape()
square_obj = Square()
triangle_obj = Triangle()

# Now pass each object, one at a time, to the get_area() function and see the
# result.
get_area(shape_obj)
get_area(square_obj)
get_area(triangle_obj)
```

Nous devrions voir cette sortie:

Aucun
4
6,0

Que se passe-t-il sans polymorphisme?
Sans polymorphisme, une vérification de type peut être requise avant d'effectuer une action sur un objet pour déterminer la méthode correcte à appeler. L' **exemple de compteur** suivant exécute la même tâche que le code précédent, mais sans l'utilisation du polymorphisme, la fonction `get_area()` doit faire plus de travail.

```
class Square:

    side_length = 2

    def calculate_square_area(self):
        return self.side_length ** 2

class Triangle:

    base_length = 4
    height = 3

    def calculate_triangle_area(self):
        return (0.5 * self.base_length) * self.height

def get_area(input_obj):

    # Notice the type checks that are now necessary here. These type checks
    # could get very complicated for a more complex example, resulting in
    # duplicate and difficult to maintain code.
```

```
    if type(input_obj).__name__ == "Square":
        area = input_obj.calculate_square_area()

    elif type(input_obj).__name__ == "Triangle":
        area = input_obj.calculate_triangle_area()

    print(area)

# Create one object of each class
square_obj = Square()
triangle_obj = Triangle()

# Now pass each object, one at a time, to the get_area() function and see the
# result.
get_area(square_obj)
get_area(triangle_obj)
```

Nous devrions voir cette sortie:

4
6,0

Note importante
Notez que les classes utilisées dans l'exemple du compteur sont des classes "new style" et héritent implicitement de la classe d'objet si Python 3 est utilisé. Le polymorphisme fonctionnera à la fois dans Python 2.x et 3.x, mais le code du contre-exemple du polymorphisme générera une exception s'il est exécuté dans un interpréteur Python 2.x en raison du type (input_obj). **name** renverra "instance" au lieu du nom de la classe si elles n'héritent pas explicitement de l'objet, ce qui entraîne que la zone n'est jamais affectée.

Duck Typing

Polymorphisme sans héritage sous forme de dactylographie comme disponible en Python grâce à son système de typage dynamique. Cela signifie que tant que les classes contiennent les mêmes méthodes, l'interpréteur Python ne les distingue pas, car la seule vérification des appels a lieu au moment de l'exécution.

```
class Duck:
    def quack(self):
        print("Quaaaaaack!")
    def feathers(self):
        print("The duck has white and gray feathers.")

class Person:
    def quack(self):
        print("The person imitates a duck.")
    def feathers(self):
        print("The person takes a feather from the ground and shows it.")
    def name(self):
        print("John Smith")

def in_the_forest(obj):
    obj.quack()
    obj.feathers()
```

```
donald = Duck()
john = Person()
in_the_forest(donald)
in_the_forest(john)
```

La sortie est la suivante:

> Quaaaaaack!
> Le canard a des plumes blanches et grises.
> La personne imite un canard.
> La personne prend une plume du sol et la montre.

Lire Polymorphisme en ligne: https://riptutorial.com/fr/python/topic/5100/polymorphisme

Chapitre 152: Portée variable et liaison

Syntaxe

- global a, b, c
- non local a, b
- x = quelque chose # lie x
- (x, y) = quelque chose # lie x et y
- x + = quelque chose # lie x. De même pour tous les autres "op ="
- del x # lie x
- pour x dans quelque chose: # lie x
- avec quelque chose comme x: # lie x
- sauf exception comme ex: # lie ex à l'intérieur du bloc

Examples

Variables globales

Dans Python, les variables à l'intérieur des fonctions sont considérées comme locales si et seulement si elles apparaissent dans la partie gauche d'une instruction d'affectation ou dans une autre occurrence de liaison. sinon, une telle liaison est recherchée dans des fonctions englobantes, jusqu'au niveau global. Cela est vrai même si l'instruction d'affectation n'est jamais exécutée.

```
x = 'Hi'

def read_x():
    print(x)    # x is just referenced, therefore assumed global

read_x()        # prints Hi

def read_y():
    print(y)    # here y is just referenced, therefore assumed global

read_y()        # NameError: global name 'y' is not defined

def read_y():
    y = 'Hey'   # y appears in an assignment, therefore it's local
    print(y)    # will find the local y

read_y()        # prints Hey

def read_x_local_fail():
    if False:
        x = 'Hey'  # x appears in an assignment, therefore it's local
    print(x)    # will look for the _local_ z, which is not assigned, and will not be found

read_x_local_fail()   # UnboundLocalError: local variable 'x' referenced before assignment
```

Normalement, une affectation à l'intérieur d'une étendue ombrera les variables externes du même

nom:

```
x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
change_local_x()  # prints Bye
print(x)  # prints Hi
```

Déclarer un nom `global` signifie que, pour le reste de la portée, toute affectation au nom se produira au niveau supérieur du module:

```
x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # prints Bye
print(x)  # prints Bye
```

Le mot-clé `global` signifie que les affectations auront lieu au niveau supérieur du module, et non au niveau supérieur du programme. Les autres modules nécessiteront toujours l'accès habituel aux variables du module.

En résumé: pour savoir si une variable `x` est locale à une fonction, vous devez lire l' *intégralité de la* fonction:

1. si vous avez trouvé `global x` , alors `x` est une variable **globale**
2. Si vous avez trouvé un `nonlocal x` , alors `x` appartient à une fonction englobante et n'est ni local ni global
3. Si vous avez trouvé `x = 5` ou `for x in range(3)` ou une autre liaison, alors `x` est une variable **locale**
4. Sinon, `x` appartient à une étendue englobante (étendue de la fonction, étendue globale ou fonctions intégrées).

Variables locales

Si un nom est *lié à l'* intérieur d'une fonction, il est accessible par défaut uniquement dans la fonction:

```
def foo():
    a = 5
    print(a) # ok

print(a) #  NameError: name 'a' is not defined
```

Les constructions de flux de contrôle n'ont aucun impact sur la portée (à l'exception de `except`), mais l'accès à la variable qui n'a pas encore été attribué est une erreur:

```
def foo():
    if True:
        a = 5
    print(a) # ok

b = 3
def bar():
    if False:
        b = 5
    print(b) # UnboundLocalError: local variable 'b' referenced before assignment
```

Les opérations de liaison communes sont des affectations, `for` boucles et des affectations augmentées telles que `a += 5`

Variables non locales

Python 3.x 3.0

Python 3 a ajouté un nouveau mot-clé appelé **nonlocal** . Le mot clé nonlocal ajoute un remplacement de portée à la portée interne. Vous pouvez tout lire à ce sujet dans PEP 3104 . Ceci est mieux illustré avec quelques exemples de code. L'un des exemples les plus courants est de créer une fonction pouvant s'incrémenter:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

Si vous essayez d'exécuter ce code, vous recevrez une **erreur UnboundLocalError** car la variable **num** est référencée avant d'être affectée dans la fonction la plus interne. Ajoutons nonlocal au mixage:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

Fondamentalement, `nonlocal` - `nonlocal` vous permettra d'attribuer des variables dans une portée externe, mais pas une portée globale. Donc, vous ne pouvez pas utiliser `nonlocal` dans notre fonction de `counter` , car cela tenterait d'attribuer une portée globale. Essayez-le et vous obtiendrez rapidement une `SyntaxError` . Au lieu de cela, vous devez utiliser `nonlocal` dans une fonction imbriquée.

(Notez que la fonctionnalité présentée ici est mieux implémentée en utilisant des générateurs.)

Occurrence de liaison

```
x = 5
x += 7
for x in iterable: pass
```

Chacune des instructions ci-dessus est une *occurrence de liaison* - `x` devient lié à l'objet désigné par `5` . Si cette instruction apparaît dans une fonction, alors `x` sera la fonction locale par défaut. Voir la section "Syntaxe" pour une liste des instructions de liaison.

Les fonctions ignorent la portée de la classe lors de la recherche de noms

Les classes ont une portée locale lors de la définition, mais les fonctions de la classe n'utilisent pas cette étendue lors de la recherche de noms. Comme les lambdas sont des fonctions et que les compréhensions sont implémentées à l'aide de la portée des fonctions, cela peut entraîner un comportement surprenant.

```
a = 'global'

class Fred:
    a = 'class'  # class scope
    b = (a for i in range(10))  # function scope
    c = [a for i in range(10)]  # function scope
    d = a  # class scope
    e = lambda: a  # function scope
    f = lambda a=a: a  # default argument uses class scope

    @staticmethod  # or @classmethod, or regular instance method
    def g():  # function scope
        return a

print(Fred.a)  # class
print(next(Fred.b))  # global
print(Fred.c[0])  # class in Python 2, global in Python 3
print(Fred.d)  # class
print(Fred.e())  # global
print(Fred.f())  # class
print(Fred.g()) # global
```

Les utilisateurs peu familiarisés avec le fonctionnement de cette étendue peuvent s'attendre à ce que `b` , `c` et `e` impriment la `class` .

De PEP 227 :

> Les noms dans la portée de la classe ne sont pas accessibles. Les noms sont résolus dans la portée de la fonction la plus interne. Si une définition de classe se produit dans une chaîne de portées imbriquées, le processus de résolution ignore les définitions de classe.

De la documentation de Python sur la dénomination et la liaison :

> La portée des noms définis dans un bloc de classe est limitée au bloc de classe; il ne s'étend pas aux blocs de code des méthodes - cela inclut les expressions de compréhension et de générateur car elles sont implémentées en utilisant une étendue de fonction. Cela signifie que les éléments suivants échoueront:
>
> ```
> class A:
>     a = 42
>     b = list(a + i for i in range(10))
> ```

Cet exemple utilise les références de cette réponse de Martijn Pieters, qui contient une analyse plus approfondie de ce comportement.

La commande del

Cette commande a plusieurs formes liées mais distinctes.

`del v`

Si `v` est une variable, la commande `del v` supprime la variable de son étendue. Par exemple:

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'f' is not defined
```

> Notez que `del` est une *occurrence de liaison* , ce qui signifie que, sauf indication contraire explicite (en utilisant des `nonlocal` ou `global`), `del v` affectera `v` local à la portée actuelle. Si vous envisagez de supprimer `v` dans une étendue externe, utilisez `nonlocal v` ou `global v` dans la même portée de l'instruction `del v` .

Dans tout ce qui suit, l'intention d'une commande est un comportement par défaut mais n'est pas appliquée par le langage. Une classe peut être écrite de manière à invalider cette intention.

`del v.name`

Cette commande déclenche un appel à la `v.__delattr__(name)` .

L'intention est de rendre le `name` attribut indisponible. Par exemple:

```
class A:
    pass

a = A()
a.x = 7
print(a.x) # out: 7
del a.x
print(a.x) # error: AttributeError: 'A' object has no attribute 'x'
```

`del v[item]`

Cette commande déclenche un appel à `v.__delitem__(item)` .

L'intention est que cet `item` n'appartienne pas au mappage implémenté par l'objet `v` . Par exemple:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) #  out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
```

`del v[a:b]`

Cela appelle réellement `v.__delslice__(a, b)` .

L'intention est similaire à celle décrite ci-dessus, mais avec des tranches - des plages d'éléments au lieu d'un seul élément. Par exemple:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) #  out: [0, 3, 4]
```

Voir aussi Garbage Collection # La commande del

Portée locale vs globale

Quelle est la portée locale et globale?

Toutes les variables Python accessibles à un moment donné dans le code sont de *portée locale* ou *globale* .

L'explication est que la portée locale inclut toutes les variables définies dans la fonction en cours et que la portée globale inclut les variables définies en dehors de la fonction en cours.

```
foo = 1  # global

def func():
    bar = 2  # local
    print(foo)  # prints variable foo from global scope
    print(bar)  # prints variable bar from local scope
```

On peut inspecter quelles variables sont dans quelle portée. Les fonctions intégrées `locals()` et `globals()` renvoient l'ensemble des étendues en tant que dictionnaires.

```
foo = 1

def func():
    bar = 2
    print(globals().keys())  # prints all variable names in global scope
    print(locals().keys())  # prints all variable names in local scope
```

Que se passe-t-il avec les conflits de noms?

```
foo = 1

def func():
    foo = 2  # creates a new variable foo in local scope, global foo is not affected

    print(foo)  # prints 2

    # global variable foo still exists, unchanged:
    print(globals()['foo'])  # prints 1
    print(locals()['foo'])  # prints 2
```

Pour modifier une variable globale, utilisez le mot-clé `global` :

```
foo = 1

def func():
    global foo
    foo = 2  # this modifies the global foo, rather than creating a local variable
```

La portée est définie pour tout le corps de la fonction!

Cela signifie qu'une variable ne sera jamais globale pour la moitié de la fonction et locale par la suite, ou inversement.

```
foo = 1

def func():
    # This function has a local variable foo, because it is defined down below.
    # So, foo is local from this point. Global foo is hidden.

    print(foo) # raises UnboundLocalError, because local foo is not yet initialized
    foo = 7
    print(foo)
```

De même, l'opposite:

```
foo = 1

def func():
    # In this function, foo is a global variable from the begining

    foo = 7  # global foo is modified

    print(foo)  # 7
    print(globals()['foo'])  # 7

    global foo  # this could be anywhere within the function
    print(foo)  # 7
```

Fonctions dans les fonctions

Il peut exister de nombreux niveaux de fonctions imbriquées dans les fonctions, mais au sein d'une même fonction, il n'ya qu'une seule portée locale pour cette fonction et la portée globale. Il n'y a pas de portées intermédiaires.

```
foo = 1

def f1():
    bar = 1

    def f2():
        baz = 2
        # here, foo is a global variable, baz is a local variable
        # bar is not in either scope
        print(locals().keys())  # ['baz']
        print('bar' in locals())  # False
        print('bar' in globals())  # False

    def f3():
        baz = 3
        print(bar)  # bar from f1 is referenced so it enters local scope of f3 (closure)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False

    def f4():
        bar = 4  # a new local bar which hides bar from local scope of f1
        baz = 4
        print(bar)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False
```

`global` vs `nonlocal` (Python 3 uniquement)

Ces deux mots-clés sont utilisés pour obtenir un accès en écriture aux variables qui ne sont pas locales pour les fonctions en cours.

Le mot-clé `global` déclare qu'un nom doit être traité comme une variable globale.

```
foo = 0  # global foo

def f1():
    foo = 1  # a new foo local in f1

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            foo = 3  # a new foo local in f3
            print(foo)  # 3
            foo = 30  # modifies local foo in f3 only

        def f4():
            global foo
            print(foo)  # 0
            foo = 100  # modifies global foo
```

D'un autre côté, `nonlocal` (voir Variables `nonlocal`), disponible dans Python 3, prend une variable *locale* d'une portée englobante dans la portée locale de la fonction actuelle.

De la documentation Python sur `nonlocal` :

> L'instruction nonlocal fait que les identificateurs répertoriés font référence aux variables précédemment liées dans la portée englobante la plus proche, à l'exception des globales.

Python 3.x 3.0

```
def f1():

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            nonlocal foo  # foo from f2, which is the nearest enclosing scope
            print(foo)  # 2
            foo = 20  # modifies foo from f2!
```

Lire Portée variable et liaison en ligne: https://riptutorial.com/fr/python/topic/263/portee-variable-et-liaison

Chapitre 153: PostgreSQL

Examples

Commencer

PostgreSQL est une base de données open source développée et mature. En utilisant le module `psycopg2` , nous pouvons exécuter des requêtes sur la base de données.

Installation en utilisant pip

```
pip install psycopg2
```

Utilisation de base

Supposons que nous avons une table `my_table` dans la base de données `my_database` définie comme suit.

id	Prénom	nom de famille
1	John	Biche

Nous pouvons utiliser le module `psycopg2` pour exécuter des requêtes sur la base de données de la manière suivante.

```
import psycopg2

# Establish a connection to the existing database 'my_database' using
# the user 'my_user' with password 'my_password'
con = psycopg2.connect("host=localhost dbname=my_database user=my_user password=my_password")

# Create a cursor
cur = con.cursor()

# Insert a record into 'my_table'
cur.execute("INSERT INTO my_table(id, first_name, last_name) VALUES (2, 'Jane', 'Doe');")

# Commit the current transaction
con.commit()

# Retrieve all records from 'my_table'
cur.execute("SELECT * FROM my_table;")
results = cur.fetchall()

# Close the database connection
con.close()

# Print the results
print(results)
```

```
# OUTPUT: [(1, 'John', 'Doe'), (2, 'Jane', 'Doe')]
```

Lire PostgreSQL en ligne: https://riptutorial.com/fr/python/topic/3374/postgresql

Chapitre 154: Priorité de l'opérateur

Introduction

Les opérateurs Python ont un **ordre de priorité** défini, qui détermine quels opérateurs sont évalués en premier dans une expression potentiellement ambiguë. Par exemple, dans l'expression 3 * 2 + 7, le premier 3 est multiplié par 2, puis le résultat est ajouté à 7, ce qui donne 13. L'expression n'est pas évaluée dans l'autre sens, car * a une priorité supérieure à +.

Vous trouverez ci-dessous une liste des opérateurs par ordre de priorité et une brève description de ce qu'ils font (généralement).

Remarques

De la documentation Python:

> Le tableau suivant résume les priorités de l'opérateur en Python, de la plus faible priorité (la plus faible liaison) à la plus haute priorité (la plus contraignante). Les opérateurs dans la même boîte ont la même priorité. A moins que la syntaxe ne soit explicitement donnée, les opérateurs sont binaires. Les opérateurs du même groupe de boîtes de gauche à droite (sauf pour les comparaisons, y compris les tests, qui ont tous la même priorité et la chaîne de gauche à droite et d'exponentiation, les groupes de droite à gauche).

Opérateur	La description
lambda	Expression lambda
sinon	Expression conditionnelle
ou	Booléen OU
et	Booléen ET
pas x	Booléen PAS
in, not in, is, n'est pas, <, <=,>,> =, <>,! =, ==	Comparaisons, y compris les tests d'adhésion et les tests d'identité
|	Bit à bit OU
^	Bit par bit XOR
Et	Bitwise AND
<<, >>	Quarts de travail

Opérateur	La description
+, -	Addition et soustraction
*, /, //,%	Multiplication, division, reste [8]
+ x, -x, ~ x	Positif, négatif, pas binaire
**	Exponentiation [9]
x [index], x [index: index], x (arguments ...), x.attribute	Abonnement, découpage, appel, référence d'attribut
(expressions ...), [expressions ...], {key: value ...}, expressions ...	Reliure ou affichage de tuple, affichage de liste, affichage de dictionnaire, conversion de chaîne

Examples

Exemples de priorité d'opérateur simple en python.

Python suit la règle PEMDAS. PEMDAS signifie parenthèses, exposants, multiplication et division, et additions et soustractions.

Exemple:

```
>>> a, b, c, d = 2, 3, 5, 7
>>> a ** (b + c)  # parentheses
256
>>> a * b ** c  # exponent: same as `a * (b ** c)`
7776
>>> a + b * c / d  # multiplication / division: same as `a + (b * c / d)`
4.142857142857142
```

Extras: les règles mathématiques tiennent, mais pas toujours :

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

Lire Priorité de l'opérateur en ligne: https://riptutorial.com/fr/python/topic/5040/priorite-de-l-operateur

Chapitre 155: Processus et threads

Introduction

La plupart des programmes sont exécutés ligne par ligne, exécutant un seul processus à la fois. Les threads permettent à plusieurs processus de circuler indépendamment les uns des autres. Le threading avec plusieurs processeurs permet aux programmes d'exécuter plusieurs processus simultanément. Cette rubrique documente l'implémentation et l'utilisation des threads en Python.

Examples

Global Interpreter Lock

Les performances du multithreading Python peuvent souvent être affectées par le **Global Interpreter Lock** . En bref, même si vous pouvez avoir plusieurs threads dans un programme Python, une seule instruction de bytecode peut s'exécuter en parallèle à tout moment, quel que soit le nombre de processeurs.

En tant que tel, le multithreading dans les cas où les opérations sont bloquées par des événements externes - comme l'accès au réseau - peut être très efficace:

```
import threading
import time


def process():
    time.sleep(2)


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.00s
# Out: Four runs took 2.00s
```

Notez que même si chaque `process` pris 2 secondes pour s'exécuter, les quatre processus ensemble ont pu être exécutés efficacement en parallèle, en prenant 2 secondes au total.

Cependant, le multithreading dans les cas où des calculs intensifs sont effectués en code Python - comme beaucoup de calculs - ne se traduit pas par une amélioration considérable et peut même

être plus lent que l'exécution en parallèle:

```
import threading
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.05s
# Out: Four runs took 14.42s
```

Dans ce dernier cas, le multitraitement peut être efficace car plusieurs processus peuvent, bien entendu, exécuter plusieurs instructions simultanément:

```
import multiprocessing
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))
```

```
start = time.time()
processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.07s
# Out: Four runs took 2.30s
```

Exécution dans plusieurs threads

Utilisez `threading.Thread` pour exécuter une fonction dans un autre thread.

```
import threading
import os

def process():
    print("Pid is %s, thread id is %s" % (os.getpid(), threading.current_thread().name))

threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()

# Out: Pid is 11240, thread id is Thread-1
# Out: Pid is 11240, thread id is Thread-2
# Out: Pid is 11240, thread id is Thread-3
# Out: Pid is 11240, thread id is Thread-4
```

Exécution dans plusieurs processus

Utilisez `multiprocessing.Process` pour exécuter une fonction dans un autre processus. L'interface est similaire à `threading.Thread` :

```
import multiprocessing
import os

def process():
    print("Pid is %s" % (os.getpid(),))

processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()

# Out: Pid is 11206
# Out: Pid is 11207
# Out: Pid is 11208
# Out: Pid is 11209
```

État de partage entre les threads

Comme tous les threads s'exécutent dans le même processus, tous les threads ont accès aux mêmes données.

Toutefois, l'accès simultané aux données partagées doit être protégé par un verrou pour éviter les problèmes de synchronisation.

```
import threading

obj = {}
obj_lock = threading.Lock()

def objify(key, val):
    print("Obj has %d values" % len(obj))
    with obj_lock:
        obj[key] = val
    print("Obj now has %d values" % len(obj))

ts = [threading.Thread(target=objify, args=(str(n), n)) for n in range(4)]
for t in ts:
    t.start()
for t in ts:
    t.join()
print("Obj final result:")
import pprint; pprint.pprint(obj)

# Out: Obj has 0 values
# Out:  Obj has 0 values
# Out: Obj now has 1 values
# Out: Obj now has 2 valuesObj has 2 values
# Out: Obj now has 3 values
# Out:
# Out:  Obj has 3 values
# Out: Obj now has 4 values
# Out: Obj final result:
# Out: {'0': 0, '1': 1, '2': 2, '3': 3}
```

État de partage entre les processus

Le code exécuté dans différents processus ne partage pas, par défaut, les mêmes données. Cependant, le module de `multiprocessing` contient des primitives permettant de partager des valeurs entre plusieurs processus.

```
import multiprocessing

plain_num = 0
shared_num = multiprocessing.Value('d', 0)
lock = multiprocessing.Lock()

def increment():
    global plain_num
    with lock:
        # ordinary variable modifications are not visible across processes
        plain_num += 1
        # multiprocessing.Value modifications are
```

```
        shared_num.value += 1

ps = [multiprocessing.Process(target=increment) for n in range(4)]
for p in ps:
    p.start()
for p in ps:
    p.join()

print("plain_num is %d, shared_num is %d" % (plain_num, shared_num.value))

# Out: plain_num is 0, shared_num is 4
```

Lire Processus et threads en ligne: https://riptutorial.com/fr/python/topic/4110/processus-et-threads

Chapitre 156: Profilage

Examples

%% timeit et% timeit dans IPython

Concaténation de chaîne de profilage:

```
In [1]: import string

In [2]: %%timeit s=""; long_list=list(string.ascii_letters)*50
  ....: for substring in long_list:
  ....:   s+=substring
  ....:
1000 loops, best of 3: 570 us per loop

In [3]: %%timeit long_list=list(string.ascii_letters)*50
  ....: s="".join(long_list)
  ....:
100000 loops, best of 3: 16.1 us per loop
```

Boucles de profilage sur les itérables et les listes:

```
In [4]: %timeit for i in range(100000):pass
100 loops, best of 3: 2.82 ms per loop

In [5]: %timeit for i in list(range(100000)):pass
100 loops, best of 3: 3.95 ms per loop
```

fonction timeit ()

Répétition du profil d'éléments dans un tableau

```
>>> import timeit
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

ligne de commande timeit

Conciliation des nombres

```
python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 3: 29.2 usec per loop

python -m timeit "'-'.join(map(str,range(100)))"
100000 loops, best of 3: 19.4 usec per loop
```

line_profiler en ligne de commande

Le code source avec la directive @profile avant la fonction que nous voulons profiler:

```
import requests

@profile
def slow_func():
    s = requests.session()
    html=s.get("https://en.wikipedia.org/").text
    sum([pow(ord(x),3.1) for x in list(html)])

for i in range(50):
    slow_func()
```

Utilisation de la commande kernprof pour calculer ligne par ligne

```
$ kernprof -lv so6.py

Wrote profile results to so6.py.lprof
Timer unit: 4.27654e-07 s

Total time: 22.6427 s
File: so6.py
Function: slow_func at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def slow_func():
     6        50        20729    414.6      0.0      s = requests.session()
     7        50     47618627 952372.5     89.9
html=s.get("https://en.wikipedia.org/").text
     8        50      5306958 106139.2     10.0      sum([pow(ord(x),3.1) for x in
list(html)])
```

La demande de page est presque toujours plus lente que tout calcul basé sur les informations de la page.

Utilisation de cProfile (Preferred Profiler)

Python inclut un profileur appelé cProfile. Ceci est généralement préférable à l'utilisation de timeit.

Il décompose tout votre script et pour chaque méthode dans votre script, il vous dit:

- `ncalls` : Le nombre de fois qu'une méthode a été appelée
- `tottime` : Temps total passé dans la fonction donnée (hors temps passé dans les appels aux sous-fonctions)
- `percall` : temps passé par appel. Ou le quotient de tottime divisé par les appels
- `cumtime` : Le temps cumulé passé dans cette sous-fonction et toutes les sous-fonctions (de l'invocation à la sortie). Ce chiffre est précis même pour les fonctions récursives.
- `percall` : est le quotient de cumtime divisé par des appels primitifs
- `filename:lineno(function)` : fournit les données respectives de chaque fonction

Le cProfiler peut être facilement appelé sur la ligne de commande en utilisant:

```
$ python -m cProfile main.py
```

Pour trier la liste des méthodes profilées renvoyées en fonction du temps écoulé dans la méthode:

```
$ python -m cProfile -s time main.py
```

Lire Profilage en ligne: https://riptutorial.com/fr/python/topic/3818/profilage

Chapitre 157: Programmation fonctionnelle en Python

Introduction

La programmation fonctionnelle décompose un problème en un ensemble de fonctions. Idéalement, les fonctions ne prennent que des entrées et produisent des sorties, et n'ont pas d'état interne qui affecte la sortie produite pour une entrée donnée.

Examples

Fonction Lambda

Une fonction anonyme, définie avec lambda. Les paramètres du lambda sont définis à gauche du côlon. Le corps de la fonction est défini à droite du côlon. Le résultat de l'exécution du corps de la fonction est (implicitement) renvoyé.

```
s=lambda x:x*x
s(2)     =>4
```

Fonction de la carte

Map prend une fonction et une collection d'éléments. Il crée une nouvelle collection vide, exécute la fonction sur chaque élément de la collection d'origine et insère chaque valeur de retour dans la nouvelle collection. Il renvoie la nouvelle collection.

Ceci est une carte simple qui prend une liste de noms et retourne une liste des longueurs de ces noms:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)     =>[4, 4, 3]
```

Réduire la fonction

Réduire prend une fonction et une collection d'éléments. Il renvoie une valeur créée en combinant les éléments.

C'est une simple réduction. Il renvoie la somme de tous les éléments de la collection.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)     =>10
```

Fonction de filtre

Le filtre prend une fonction et une collection. Il renvoie une collection de chaque élément pour lequel la fonction a renvoyé True.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]    # outputs[5,6]
```

Lire Programmation fonctionnelle en Python en ligne:
https://riptutorial.com/fr/python/topic/9552/programmation-fonctionnelle-en-python

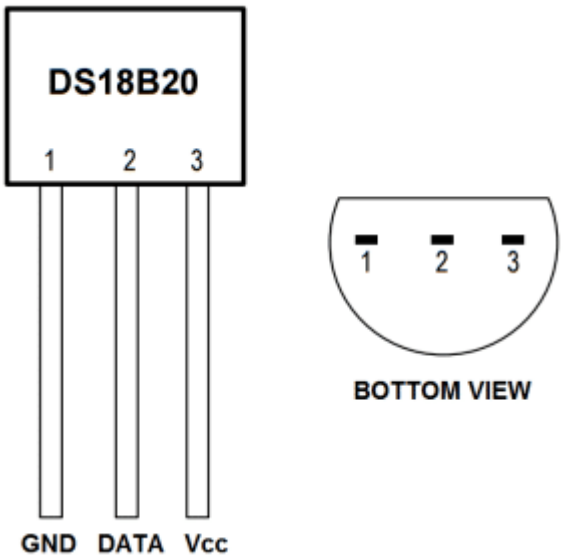
Chapitre 158: Programmation IoT avec Python et Raspberry PI

Examples

Exemple - Capteur de température

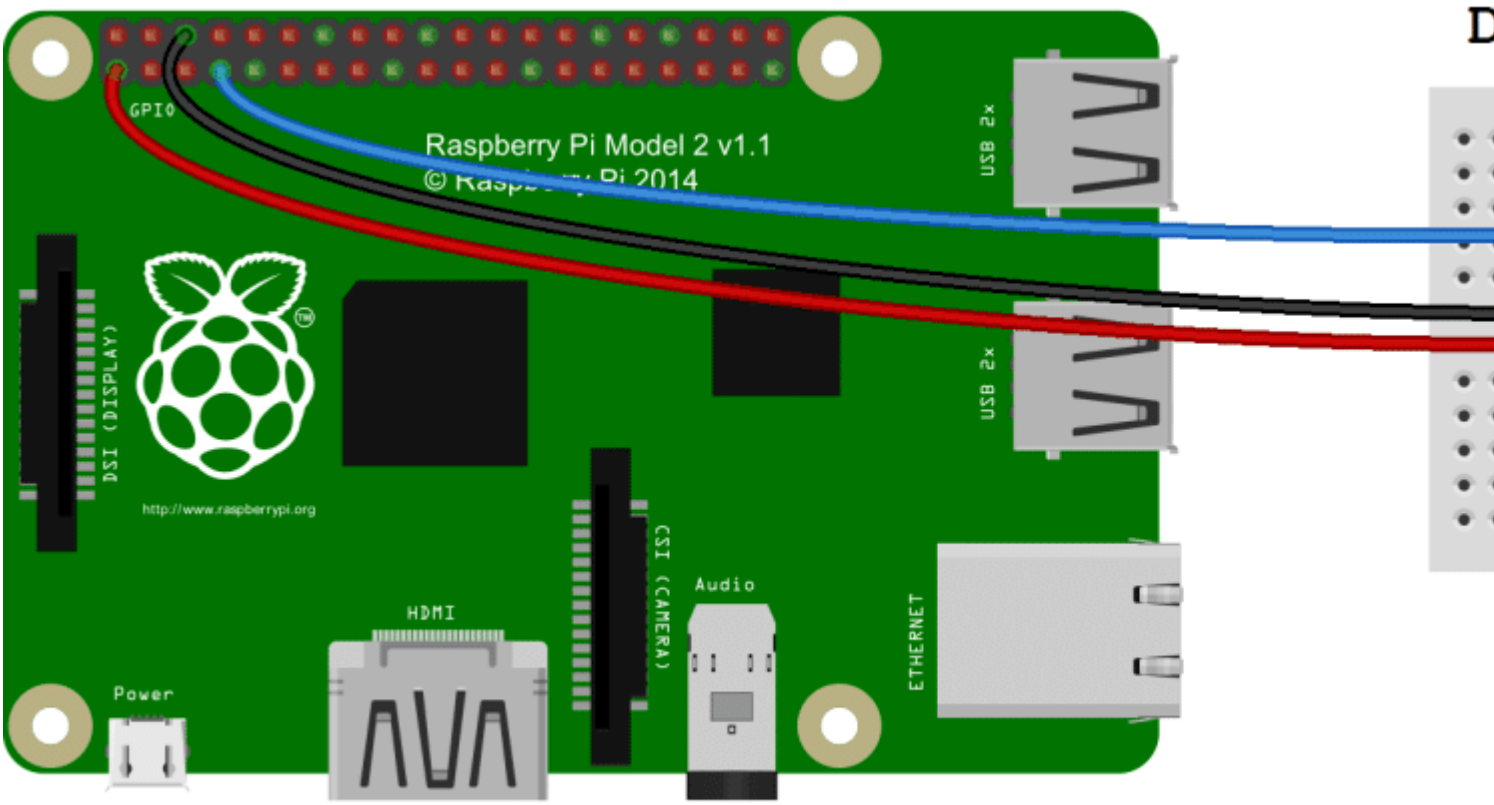
Interfaçage de DS18B20 avec Raspberry Pi

Connexion de DS18B20 avec Raspberry Pi



Vous pouvez voir qu'il y a trois terminaux

1. Vcc
2. Gnd
3. Données (protocole à un fil)



R1 est une résistance de 4,7k ohms pour augmenter le niveau de tension

1. **Vcc** doit être connecté à l'une des broches 5v ou 3.3v de Raspberry pi (PIN: 01, 02, 04, 17).
2. **Gnd** devrait être connecté à n'importe quelle punaise de Raspberry pi (PIN: 06, 09, 14, 20, 25).
3. **DATA** doit être connecté à (PIN: 07)

Activer l'interface à un fil du côté RPi

4. Connectez-vous à Raspberry Pi en utilisant un mastic ou tout autre terminal Linux / Unix.
5. Après la connexion, ouvrez le fichier /boot/config.txt dans votre navigateur préféré.

   nano /boot/config.txt
6. Ajoutez maintenant la ligne `dtoverlay=w1-gpio` à la fin du fichier.
7. Maintenant, redémarrez le Raspberry Pi `sudo reboot` .
8. Connectez-vous à Raspberry pi et exécutez `sudo modprobe g1-gpio`
9. Ensuite, lancez `sudo modprobe w1-therm`
10. Maintenant, allez dans le répertoire / sys / bus / w1 / devices `cd /sys/bus/w1/devices`
11. Vous allez maintenant découvrir un répertoire virtuel créé à partir de votre capteur de température à partir du 28 - ********.

12. Allez dans ce répertoire `cd 28-********`

13. Maintenant, il y a un nom de fichier **w1-slave** , Ce fichier contient la température et d'autres informations comme le CRC. `cat w1-slave` .

Maintenant, écrivez un module en python pour lire la température

```
import glob
import time

RATE = 30
sensor_dirs = glob.glob("/sys/bus/w1/devices/28*")

if len(sensor_dirs) != 0:
    while True:
        time.sleep(RATE)
        for directories in sensor_dirs:
            temperature_file = open(directories + "/w1_slave")
            # Reading the files
            text = temperature_file.read()
            temperature_file.close()
            # Split the text with new lines (\n) and select the second line.
            second_line = text.split("\n")[1]
            # Split the line into words, and select the 10th word
            temperature_data = second_line.split(" ")[9]
            # We will read after ignoring first two character.
            temperature = float(temperature_data[2:])
            # Now normalise the temperature by dividing 1000.
            temperature = temperature / 1000
            print 'Address : '+str(directories.split('/')[-1])+', Temperature :
'+str(temperature)
```

Le module ci-dessus python imprimera la température en fonction de l'adresse pour une durée infinie. Le paramètre RATE est défini pour modifier ou ajuster la fréquence de la requête de température du capteur.

Diagramme de broche GPIO

1. [https://www.element14.com/community/servlet/JiveServlet/previewBody/73950-102-11-339300/pi3_gpio.png][3]

Lire Programmation IoT avec Python et Raspberry PI en ligne:
https://riptutorial.com/fr/python/topic/10735/programmation-iot-avec-python-et-raspberry-pi

Chapitre 159: py.test

Examples

Mise en place de py.test

`py.test` est l'une des bibliothèques de test tierces disponibles pour Python. Il peut être installé en utilisant `pip` with

```
pip install pytest
```

Le code à tester

Disons que nous testons une fonction d'ajout dans `projectroot/module/code.py` :

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

Le code de test

Nous créons un fichier de test dans `projectroot/tests/test_code.py` . Le fichier **doit commencer par `test_`** pour être reconnu en tant que fichier de test.

```
# projectroot/tests/test_code.py
from module import code


def test_add():
    assert code.add(1, 2) == 3
```

Lancer le test

À partir de `projectroot` nous `projectroot` simplement `py.test` :

```
# ensure we have the modules
$ touch tests/__init__.py
$ touch module/__init__.py
$ py.test
=========================================================== test session starts
============================================================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py .
```

```
================================================ 1 passed in 0.01 seconds
================================================
```

Essais défaillants

Un test défaillant fournira des informations utiles sur ce qui a mal tourné:

```
# projectroot/tests/test_code.py
from module import code


def test_add__failing():
    assert code.add(10, 11) == 33
```

Résultats:

```
$ py.test
================================================== test session starts
==================================================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py F

======================================================== FAILURES
========================================================
_____________________________________________________ test_add__failing
_____________________________________________________

    def test_add__failing():
>       assert code.add(10, 11) == 33
E       assert 21 == 33
E        +  where 21 = <function add at 0x105d4d6e0>(10, 11)
E        +    where <function add at 0x105d4d6e0> = code.add

tests/test_code.py:5: AssertionError
================================================ 1 failed in 0.01 seconds
================================================
```

Introduction aux tests

Des tests plus compliqués doivent parfois être configurés avant d'exécuter le code que vous souhaitez tester. Il est possible de le faire dans la fonction de test elle-même, mais vous vous retrouvez avec de grosses fonctions de test à tel point qu'il est difficile de déterminer où la configuration s'arrête et que le test commence. Vous pouvez également obtenir beaucoup de code de configuration en double entre vos différentes fonctions de test.

Notre fichier de code:

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
```

```
        self.bar = 2
```

Notre fichier de test:

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def test_foo_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Ce sont des exemples assez simples, mais si notre objet `Stuff` nécessitait beaucoup plus de configuration, il deviendrait trop compliqué. Nous voyons qu'il y a du code dupliqué entre nos cas de test, alors commençons par le transformer en une fonction distincte.

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def get_prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff


def test_foo_updates():
    my_stuff = get_prepped_stuff()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = get_prepped_stuff()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Cela semble mieux, mais nous avons toujours l' `my_stuff = get_prepped_stuff()` encombre nos fonctions de test.

luminaires py.test à la rescousse!

Les montages sont des versions beaucoup plus puissantes et flexibles des fonctions de configuration de test. Ils peuvent faire beaucoup plus que ce que nous utilisons ici, mais nous le ferons une étape à la fois.

Nous `get_prepped_stuff` remplacer `get_prepped_stuff` par un appareil appelé `prepped_stuff` . Vous voulez nommer vos appareils avec des noms plutôt que des verbes à cause de la façon dont les appareils seront utilisés dans les fonctions de test eux-mêmes plus tard. `@pytest.fixture` indique que cette fonction spécifique doit être gérée comme une installation plutôt que comme une fonction normale.

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

Maintenant, nous devons mettre à jour les fonctions de test pour qu'elles utilisent le projecteur. Cela se fait en ajoutant un paramètre à leur définition qui correspond exactement au nom du projecteur. Lorsque py.test est exécuté, il lance le fixture avant d'exécuter le test, puis transmet la valeur de retour du fixture dans la fonction de test via ce paramètre. (Notez que les appareils n'ont pas **besoin** de retourner une valeur; ils peuvent faire d'autres choses, comme appeler une ressource externe, arranger les choses sur le système de fichiers, mettre des valeurs dans une base de données, quels que soient les tests requis)

```
def test_foo_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Maintenant, vous pouvez voir pourquoi nous l'avons nommé avec un nom. mais la ligne `my_stuff = prepped_stuff` est quasiment inutile, alors utilisons simplement `prepped_stuff` place.

```
def test_foo_updates(prepped_stuff):
    assert 1 == prepped_stuff.foo
    prepped_stuff.foo = 30000
    assert prepped_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    assert 2 == prepped_stuff.bar
    prepped_stuff.bar = 42
    assert 42 == prepped_stuff.bar
```

Maintenant, nous utilisons des appareils! Nous pouvons aller plus loin en modifiant la portée de

l’appareil (pour qu’il ne s’exécute qu’une seule fois par module d’exécution ou exécution de test suite à la fonction test), en construisant des appareils utilisant d’autres appareils, en paramétrant le les tests utilisant cet appareil sont exécutés plusieurs fois, une fois pour chaque paramètre donné à l'appareil, comme les appareils qui lisent les valeurs du module qui les appelle ... comme mentionné précédemment, les appareils ont beaucoup plus de puissance et de souplesse qu'une fonction normale.

Nettoyage après les tests sont faits.

Disons que notre code a grandi et que notre objet Stuff nécessite maintenant un nettoyage spécial.

```
# projectroot/module/stuff.py
class Stuff(object):
def prep(self):
    self.foo = 1
    self.bar = 2

def finish(self):
    self.foo = 0
    self.bar = 0
```

Nous pourrions ajouter du code pour appeler le nettoyage au bas de chaque fonction de test, mais les appareils fournissent un meilleur moyen de le faire. Si vous ajoutez une fonction au fixture et que vous l'enregistrez en tant que **finaliseur** , le code de la fonction finalizer sera appelé après le test utilisant le fixture. Si la portée de l'appareil est supérieure à une seule fonction (comme un module ou une session), le finaliseur sera exécuté une fois que tous les tests de la portée auront été effectués, donc après l'exécution du module ou à la fin de la session de test .

```
@pytest.fixture
def prepped_stuff(request):  # we need to pass in the request to use finalizers
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    def fin():  # finalizer function
        # do all the cleanup here
        my_stuff.finish()
    request.addfinalizer(fin)  # register fin() as a finalizer
    # you can do more setup here if you really want to
    return my_stuff
```

Utiliser la fonction de finaliseur à l’intérieur d’une fonction peut être un peu difficile à comprendre à première vue, surtout lorsque vous avez des appareils plus complexes. Vous pouvez plutôt utiliser un **indicateur de rendement** pour faire la même chose avec un flux d'exécution plus lisible. La seule différence réelle est qu'au lieu d'utiliser `return` nous utilisons un `yield` à la partie du montage où la configuration est effectuée et le contrôle doit aller à une fonction de test, puis ajouter tout le code de nettoyage après le `yield` . Nous la décorons également en tant que `yield_fixture` pour que py.test sache comment le gérer.

```
@pytest.yield_fixture
def prepped_stuff():  # it doesn't need request now!
    # do setup
```

```
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    # setup is done, pass control to the test functions
    yield my_stuff
    # do cleanup
    my_stuff.finish()
```

Et cela conclut l'Intro à Test Fixtures!

Pour plus d'informations, consultez la documentation officielle sur le luminaire py.test et la documentation officielle sur le rendement.

Lire py.test en ligne: https://riptutorial.com/fr/python/topic/7054/py-test

Chapitre 160: pyaudio

Introduction

PyAudio fournit des liaisons Python pour PortAudio, la bibliothèque d'E / S audio multi-plateformes. Avec PyAudio, vous pouvez facilement utiliser Python pour lire et enregistrer de l'audio sur diverses plates-formes. PyAudio est inspiré par:

1.pyPortAudio / fastaudio: liaisons Python pour l'API PortAudio v18.

2.tkSnack: toolkit audio multi-plateforme pour Tcl / Tk et Python.

Remarques

Remarque: stream_callback est appelé dans un thread séparé (à partir du thread principal). Les exceptions qui se produisent dans le stream_callback seront les suivantes:
1 .imprimez une traceback sur une erreur standard pour faciliter le débogage,
2 .queue l'exception à lancer (à un moment donné) dans le thread principal, et
3 .retendez paAbort à PortAudio pour arrêter le flux.
Remarque: N'appelez pas Stream.read () ou Stream.write () si vous utilisez une opération non bloquante.
Voir: Signature de rappel de PortAudio pour plus de détails:
http://portaudio.com/docs/v19-doxydocs/portaudio_8h.html#a8a60fb2a5ec9cbade3f54a9c978e2710

Examples

Mode de rappel E / S audio

```
"""PyAudio Example: Play a wave file (callback version)."""

import pyaudio
import wave
import time
import sys

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
p = pyaudio.PyAudio()

# define callback (2)
def callback(in_data, frame_count, time_info, status):
    data = wf.readframes(frame_count)
    return (data, pyaudio.paContinue)
```

```
# open stream using callback (3)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True,
                stream_callback=callback)

# start the stream (4)
stream.start_stream()

# wait for stream to finish (5)
while stream.is_active():
    time.sleep(0.1)

# stop stream (6)
stream.stop_stream()
stream.close()
wf.close()

# close PyAudio (7)
p.terminate()
```

En mode de rappel, PyAudio appelle une fonction de rappel spécifiée (2) chaque fois qu'il a besoin de nouvelles données audio (pour la lecture) et / ou lorsque de nouvelles données audio (enregistrées) sont disponibles. Notez que PyAudio appelle la fonction de rappel dans un thread séparé. La fonction a le `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` signature suivant `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` et doit renvoyer un tuple contenant les trames de données audio `frame_count` et un indicateur indiquant s'il y a plus d'images à lire / enregistrer.

Commencez à traiter le flux audio à l'aide de **pyaudio.Stream.start_stream ()** (4), qui appellera la fonction de rappel à plusieurs reprises jusqu'à ce que cette fonction renvoie **pyaudio.paComplete** .

Pour que le flux reste actif, le thread principal ne doit pas se terminer, par exemple en dormant (5).

Mode de blocage Audio I / O

"" "PyAudio Exemple: Lire un fichier wave." ""

```
import pyaudio
import wave
import sys

CHUNK = 1024

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
```

```
p = pyaudio.PyAudio()

# open stream (2)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True)

# read data
data = wf.readframes(CHUNK)

# play stream (3)
while len(data) > 0:
    stream.write(data)
    data = wf.readframes(CHUNK)

# stop stream (4)
stream.stop_stream()
stream.close()

# close PyAudio (5)
p.terminate()
```

Pour utiliser PyAudio, instanciez d'abord PyAudio en utilisant **pyaudio.PyAudio ()** (1), qui configure le système portaudio.

Pour enregistrer ou lire de l'audio, ouvrez un flux sur le périphérique souhaité avec les paramètres audio souhaités à l'aide de **pyaudio.PyAudio.open ()** (2). Cela configure un **pyaudio.Stream** pour lire ou enregistrer de l'audio.

Jouez l'audio en écrivant des données audio dans le flux à l'aide de **pyaudio.Stream.write ()** ou lisez les données audio du flux à l'aide de **pyaudio.Stream.read ()** . (3)

Notez qu'en *mode* " *blocage* ", chaque **pyaudio.Stream.write ()** ou **pyaudio.Stream.read () se** bloque jusqu'à ce que toutes les images données / demandées aient été lues / enregistrées. Sinon, pour générer des données audio à la volée ou traiter immédiatement les données audio enregistrées, utilisez le «mode rappel» (*reportez-vous à l'exemple du* mode rappel).

Utilisez pyaudio.Stream.stop_stream () pour suspendre la lecture / l'enregistrement et **pyaudio.Stream.close ()** pour terminer le flux. (4)

Enfin, terminez la session **portaudio en** utilisant **pyaudio.PyAudio.terminate ()** (5)

Lire pyaudio en ligne: https://riptutorial.com/fr/python/topic/10627/pyaudio

Chapitre 161: pygame

Introduction

Pygame est la bibliothèque de référence pour créer des applications multimédia, en particulier des jeux, en Python. Le site officiel est http://www.pygame.org/ .

Syntaxe

- pygame.mixer.init (fréquence = 22050, taille = -16, canaux = 2, tampon = 4096)
- pygame.mixer.pre_init (fréquence, taille, canaux, tampon)
- pygame.mixer.quit ()
- pygame.mixer.get_init ()
- pygame.mixer.stop ()
- pygame.mixer.pause ()
- pygame.mixer.unpause ()
- pygame.mixer.fadeout (heure)
- pygame.mixer.set_num_channels (count)
- pygame.mixer.get_num_channels ()
- pygame.mixer.set_reserved (count)
- pygame.mixer.find_channel (force)
- pygame.mixer.get_busy ()

Paramètres

Paramètre	Détails
compter	Un entier positif représentant quelque chose comme le nombre de canaux à réserver.
Obliger	Une valeur booléenne (`False` ou `True`) qui détermine si `find_channel()` doit renvoyer un canal (inactif ou non) avec `True` ou non (s'il n'y a pas de canaux inactifs) avec `False`

Examples

Installation de pygame

Avec `pip` :

```
pip install pygame
```

Avec `conda` :

```
conda install -c tlatorre pygame=1.9.2
```

Téléchargement direct depuis le site web: http://www.pygame.org/download.shtml

Vous pouvez trouver les installateurs appropriés pour Windows et autres systèmes d'exploitation.

Les projets peuvent également être trouvés à http://www.pygame.org/

Module de mixage de Pygame

Le module `pygame.mixer` permet de contrôler la musique utilisée dans les programmes `pygame` . A ce jour, il existe 15 fonctions différentes pour le module de `mixer` .

Initialisation

Comme pour initialiser `pygame` avec `pygame.init()` , vous devez également initialiser `pygame.mixer` .

En utilisant la première option, nous initialisons le module en utilisant les valeurs par défaut. Vous pouvez cependant remplacer ces options par défaut. En utilisant la deuxième option, nous pouvons initialiser le module en utilisant les valeurs que nous avons manuellement mises en place. Valeurs standard:

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

Pour vérifier si nous l'avons initialisé ou non, nous pouvons utiliser `pygame.mixer.get_init()` , qui renvoie `True` si c'est le cas et `False` si ce n'est pas le cas. Pour quitter / annuler l'initialisation, utilisez simplement `pygame.mixer.quit()` . Si vous souhaitez continuer à jouer des sons avec le module, vous devrez peut-être réinitialiser le module.

Actions possibles

Lorsque votre son est en cours de lecture, vous pouvez le suspendre temporairement avec `pygame.mixer.pause()` . Pour reprendre la lecture de vos sons, utilisez simplement `pygame.mixer.unpause()` . Vous pouvez également supprimer la fin du son en utilisant `pygame.mixer.fadeout()` . Il prend un argument, qui est le nombre de millisecondes qu'il faut pour terminer l'évanouissement de la musique.

Canaux

Vous pouvez jouer autant de chansons que nécessaire tant qu'il y a suffisamment de canaux ouverts pour les prendre en charge. Par défaut, il y a 8 canaux. Pour modifier le nombre de canaux, utilisez `pygame.mixer.set_num_channels()` . L'argument est un entier non négatif. Si le nombre de canaux diminue, tous les sons lus sur les canaux supprimés s'arrêtent immédiatement.

Pour connaître le nombre de canaux actuellement utilisés, appelez `pygame.mixer.get_channels(count)` . La sortie est le nombre de canaux qui ne sont pas ouverts actuellement. Vous pouvez également réserver des canaux pour les sons qui doivent être joués en utilisant `pygame.mixer.set_reserved(count)` . L'argument est également un entier non négatif. Tout son joué sur les canaux nouvellement réservés ne sera pas arrêté.

Vous pouvez également trouver quel canal n'est pas utilisé en utilisant `pygame.mixer.find_channel(force)` . Son argument est un bool: vrai ou faux. S'il n'y a aucun canal qui est inactif et que la `force` est False, il ne renverra `None` . Si la `force` est vraie, cela renverra le canal qui a joué le plus longtemps.

Lire pygame en ligne: https://riptutorial.com/fr/python/topic/8761/pygame

Chapitre 162: Pyglet

Introduction

Pyglet est un module Python utilisé pour les visuels et le son. Il n'a pas de dépendances sur les autres modules. Voir [pyglet.org] [1] pour les informations officielles. [1]: http://pyglet.org

Examples

Bonjour tout le monde à Pyglet

```
import pyglet
window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
                      font_name='Times New Roman',
                      font_size=36,
                      x=window.width//2, y=window.height//2,
                      anchor_x='center', anchor_y='center')
@window.event
def on_draw():
    window.clear()
    label.draw()
pyglet.app.run()
```

Installation de Pyglet

Installez Python, allez dans la ligne de commande et tapez:

Python 2:

```
pip install pyglet
```

Python 3:

```
pip3 install pyglet
```

Jouer du son dans Pyglet

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

Utiliser Pyglet pour OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()
```

```
@win.event()
def on_draw():
    #OpenGL goes here. Use OpenGL as normal.

pyglet.app.run()
```

Dessiner des points en utilisant Pyglet et OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()
glClear(GL_COLOR_BUFFER_BIT)

@win.event
def on_draw():
    glBegin(GL_POINTS)
    glVertex2f(x, y) #x is desired distance from left side of window, y is desired distance
from bottom of window
    #make as many vertexes as you want
    glEnd
```

Pour connecter les points, remplacez `GL_POINTS` par `GL_LINE_LOOP` .

Lire Pyglet en ligne: https://riptutorial.com/fr/python/topic/8208/pyglet

Chapitre 163: PyInstaller - Distribuer du code Python

Syntaxe

- pyinstaller [options] script [script ...] | fichier spec

Remarques

PyInstaller est un module utilisé pour regrouper des applications Python dans un seul package avec toutes les dépendances. L'utilisateur peut alors exécuter l'application de package sans interpréteur python ni modules. Il regroupe correctement de nombreux paquets majeurs tels que numpy, Django, OpenCv et autres.

Quelques points importants à retenir:

- Pyinstaller prend en charge Python 2.7 et Python 3.3+
- Pyinstaller a été testé sur Windows, Linux et Mac OS X.
- Ce n'est **PAS** un compilateur croisé. (Une application Windows ne peut pas être empaquetée sous Linux. Vous devez exécuter PyInstaller dans Windows pour regrouper une application pour Windows)

Page d' accueil Documents officiels

Examples

Installation et configuration

Pyinstaller est un package Python normal. Il peut être installé en utilisant pip:

```
pip install pyinstaller
```

Installation sous Windows
Pour Windows, pywin32 ou pypiwin32 est une condition préalable. Ce dernier est installé automatiquement lorsque pyinstaller est installé à l'aide de pip.

Installation sous Mac OS X
PyInstaller fonctionne avec le Python 2.7 par défaut fourni avec Mac OS X actuel. Si des versions ultérieures de Python doivent être utilisées ou si des paquets majeurs tels que PyQT, Numpy, Matplotlib et autres doivent être utilisés, il est recommandé de les installer en utilisant soit MacPorts ou Homebrew .

Installation depuis les archives
Si pip n'est pas disponible, téléchargez l'archive compressée depuis PyPI .

Pour tester la version de développement, téléchargez l'archive compressée à partir de la branche de *développement* de la page Téléchargements de PyInstaller .

Développez l'archive et recherchez le script `setup.py` . Exécutez `python setup.py install` avec le privilège administrateur pour installer ou mettre à niveau PyInstaller.

Vérification de l'installation
Le programme de `pyinstaller` commande doit exister sur le chemin du système pour toutes les plates-formes après une installation réussie.
Vérifiez-le en tapant `pyinstaller --version` dans la ligne de commande. Cela imprimera la version actuelle du programme de désinstallation.

Utilisation de Pyinstaller

Dans le cas d'utilisation le plus simple, accédez simplement au répertoire dans lequel se trouve votre fichier et tapez:

```
pyinstaller myfile.py
```

Pyinstaller analyse le fichier et crée:

- Un fichier **myfile.spec** dans le même répertoire que `myfile.py`
- Un dossier de **compilation** dans le même répertoire que `myfile.py`
- Un dossier **dist** dans le même répertoire que `myfile.py`
- Fichiers journaux dans le dossier de **génération**

L'application fournie peut être trouvée dans le dossier **dist**

Les options
Plusieurs options peuvent être utilisées avec pyinstaller. Une liste complète des options peut être trouvée ici .

Une fois empaquetée, votre application peut être exécutée en ouvrant «dist \ myfile \ myfile.exe».

Regrouper dans un dossier

Lorsque PyInstaller est utilisé sans aucune option pour regrouper `myscript.py` , la sortie par défaut est un seul dossier (nommé `myscript`) contenant un exécutable nommé `myscript` (`myscript.exe` dans Windows) avec toutes les dépendances nécessaires.
L'application peut être distribuée en compressant le dossier dans un fichier zip.

Un mode Dossier peut être défini explicitement à l'aide de l'option `-D` ou `--onedir`

```
pyinstaller myscript.py -D
```

Avantages:

L'un des principaux avantages du regroupement dans un seul dossier est qu'il est plus facile de

déboguer les problèmes. Si des modules ne parviennent pas à importer, il peut être vérifié en inspectant le dossier.
Un autre avantage est ressenti pendant les mises à jour. S'il y a quelques modifications dans le code mais que les dépendances utilisées sont *exactement* les mêmes, les distributeurs peuvent simplement envoyer le fichier exécutable (qui est généralement plus petit que le dossier entier).

Désavantages

Le seul inconvénient de cette méthode est que les utilisateurs doivent rechercher l'exécutable parmi un grand nombre de fichiers.
Les utilisateurs peuvent également supprimer / modifier d'autres fichiers, ce qui pourrait empêcher l'application de fonctionner correctement.

Regroupement dans un fichier unique

```
pyinstaller myscript.py -F
```

Les options pour générer un seul fichier sont `-F` ou `--onefile` . Cela regroupe le programme dans un seul fichier `myscript.exe` .

Les fichiers exécutables uniques sont plus lents que le groupe à un dossier. Ils sont également plus difficiles à déboguer.

Lire PyInstaller - Distribuer du code Python en ligne:
https://riptutorial.com/fr/python/topic/2289/pyinstaller---distribuer-du-code-python

Chapitre 164: Python et Excel

Examples

Placez les données de liste dans un fichier Excel.

```
import os, sys
from openpyxl import Workbook
from datetime import datetime

dt = datetime.now()
list_values = [["01/01/2016", "05:00:00", 3], \
               ["01/02/2016", "06:00:00", 4], \
               ["01/03/2016", "07:00:00", 5], \
               ["01/04/2016", "08:00:00", 6], \
               ["01/05/2016", "09:00:00", 7]]

# Create a Workbook on Excel:
wb = Workbook()
sheet = wb.active
sheet.title = 'data'

# Print the titles into Excel Workbook:
row = 1
sheet['A'+str(row)] = 'Date'
sheet['B'+str(row)] = 'Hour'
sheet['C'+str(row)] = 'Value'

# Populate with data
for item in list_values:
    row += 1
    sheet['A'+str(row)] = item[0]
    sheet['B'+str(row)] = item[1]
    sheet['C'+str(row)] = item[2]

# Save a file by date:
filename = 'data_' + dt.strftime("%Y%m%d_%I%M%S") + '.xlsx'
wb.save(filename)

# Open the file for the user:
os.chdir(sys.path[0])
os.system('start excel.exe "%s\\%s"' % (sys.path[0], filename, ))
```

OpenPyXL

OpenPyXL est un module permettant de manipuler et de créer des `xlsx/xlsm/xltx/xltm` en mémoire.

Manipulation et lecture d'un classeur existant:

```
import openpyxl as opx
#To change an existing wookbook we located it by referencing its path
workbook = opx.load_workbook(workbook_path)
```

`load_workbook()` contient le paramètre `read_only` , la valeur `True` va charger le classeur comme read_only, cela est utile lors de la lecture de fichiers `xlsx` plus `xlsx` :

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

Une fois que vous avez chargé le classeur en mémoire, vous pouvez accéder aux feuilles individuelles à l'aide de `workbook.sheets`

```
first_sheet = workbook.worksheets[0]
```

Si vous souhaitez spécifier le nom d'une feuille disponible, vous pouvez utiliser `workbook.get_sheet_names()` .

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

Enfin, les lignes de la feuille sont accessibles à l'aide de `sheet.rows` . Pour parcourir les lignes d'une feuille, utilisez:

```
for row in sheet.rows:
    print row[0].value
```

Comme chaque `row` dans les `rows` est une liste de `Cell` , utilisez `Cell.value` pour obtenir le contenu de la cellule.

Créer un nouveau classeur en mémoire:

```
#Calling the Workbook() function creates a new book in memory
wb = opx.Workbook()

#We can then create a new sheet in the wb
ws = wb.create_sheet('Sheet Name', 0) #0 refers to the index of the sheet order in the wb
```

Plusieurs propriétés de tabulation peuvent être modifiées via openpyxl, par exemple `tabColor` :

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

Pour enregistrer notre classeur créé, nous terminons avec:

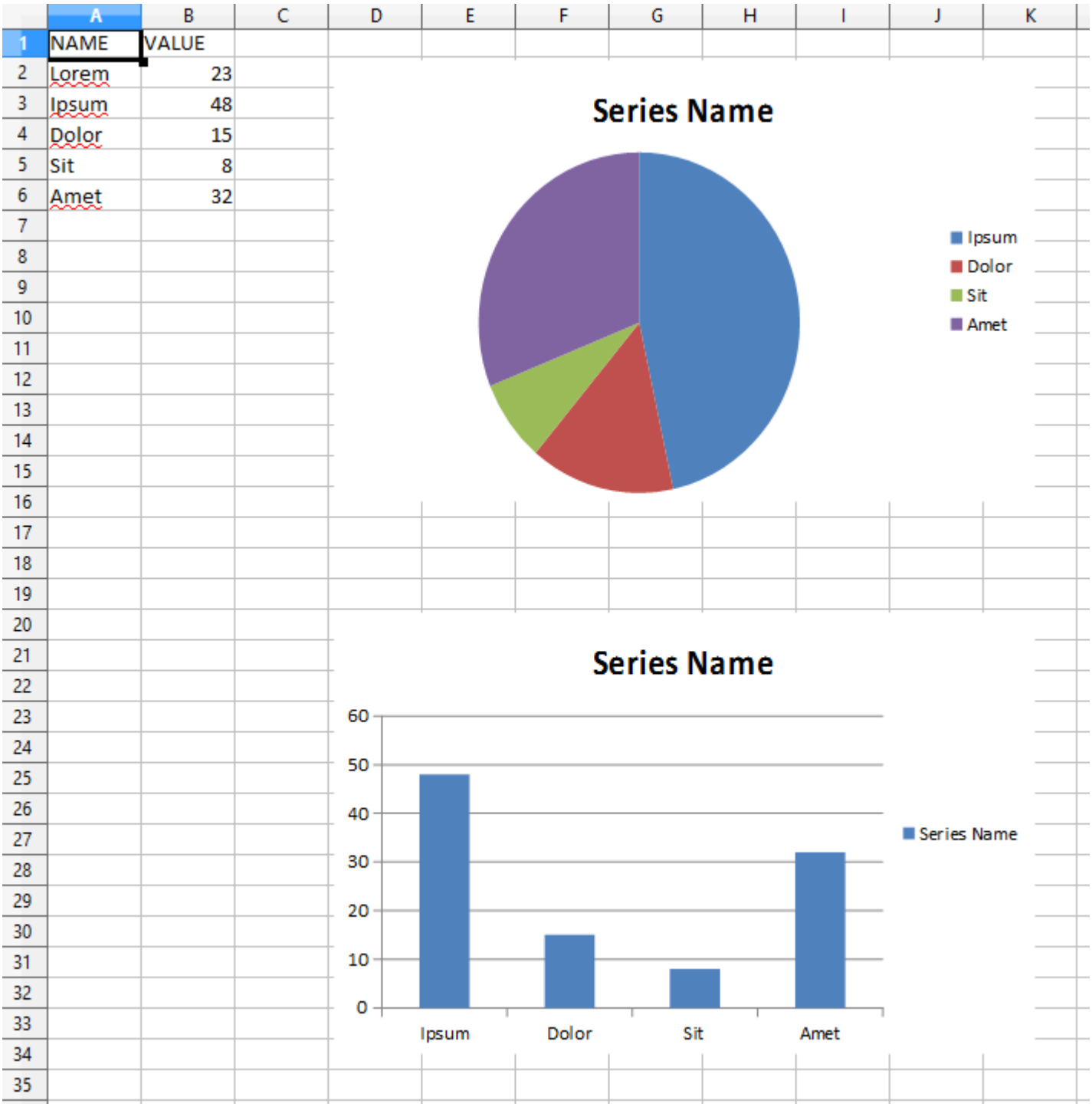
```
wb.save('filename.xlsx')
```

Créer des graphiques Excel avec xlsxwriter

```
import xlsxwriter

# sample data
chart_data = [
    {'name': 'Lorem', 'value': 23},
    {'name': 'Ipsum', 'value': 48},
    {'name': 'Dolor', 'value': 15},
```

```
    {'name': 'Sit', 'value': 8},
    {'name': 'Amet', 'value': 32}
]

# excel file path
xls_file = 'chart.xlsx'

# the workbook
workbook = xlsxwriter.Workbook(xls_file)

# add worksheet to workbook
worksheet = workbook.add_worksheet()

row_ = 0
col_ = 0

# write headers
worksheet.write(row_, col_, 'NAME')
col_ += 1
worksheet.write(row_, col_, 'VALUE')
row_ += 1

# write sample data
for item in chart_data:
    col_ = 0
    worksheet.write(row_, col_, item['name'])
    col_ += 1
    worksheet.write(row_, col_, item['value'])
    row_ += 1

# create pie chart
pie_chart = workbook.add_chart({'type': 'pie'})

# add series to pie chart
pie_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})
# insert pie chart
worksheet.insert_chart('D2', pie_chart)

# create column chart
column_chart = workbook.add_chart({'type': 'column'})

# add serie to column chart
column_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})
# insert column chart
worksheet.insert_chart('D20', column_chart)

workbook.close()
```

Résultat:



Lisez les données Excel avec le module xlrd

La bibliothèque Python xlrd consiste à extraire des données de fichiers de feuille de calcul Microsoft Excel (tm).

Installation:-

```
pip install xlrd
```

Ou vous pouvez utiliser le fichier setup.py de pypi

https://pypi.python.org/pypi/xlrd

Lire une feuille Excel: - Importez le module xlrd et ouvrez le fichier Excel avec la méthode open_workbook ().

```
import xlrd
book=xlrd.open_workbook('sample.xlsx')
```

Vérifiez le nombre de feuilles dans Excel

```
print book.nsheets
```

Imprimer les noms de feuille

```
print book.sheet_names()
```

Obtenir la feuille basée sur l'index

```
sheet=book.sheet_by_index(1)
```

Lire le contenu d'une cellule

```
cell = sheet.cell(row,col) #where row=row number and col=column number
print cell.value #to print the cell contents
```

Obtenir le nombre de lignes et le nombre de colonnes dans une feuille Excel

```
num_rows=sheet.nrows
num_col=sheet.ncols
```

Obtenez une feuille Excel par nom

```
sheets = book.sheet_names()
cur_sheet = book.sheet_by_name(sheets[0])
```

Formater des fichiers Excel avec xlsxwriter

```
import xlsxwriter

# create a new file
workbook = xlsxwriter.Workbook('your_file.xlsx')

# add some new formats to be used by the workbook
percent_format = workbook.add_format({'num_format': '0%'})
percent_with_decimal = workbook.add_format({'num_format': '0.0%'})
bold = workbook.add_format({'bold': True})
red_font = workbook.add_format({'font_color': 'red'})
remove_format = workbook.add_format()

# add a new sheet
worksheet = workbook.add_worksheet()
```

```
# set the width of column A
worksheet.set_column('A:A', 30, )

# set column B to 20 and include the percent format we created earlier
worksheet.set_column('B:B', 20, percent_format)

# remove formatting from the first row (change in height=None)
worksheet.set_row('0:0', None, remove_format)

workbook.close()
```

Lire Python et Excel en ligne: https://riptutorial.com/fr/python/topic/2986/python-et-excel

Chapitre 165: Python Lex-Yacc

Introduction

PLY est une implémentation pure-Python des populaires outils de construction du compilateur lex et yacc.

Remarques

Liens supplémentaires:

1. Documents officiels
2. Github

Examples

Premiers pas avec PLY

Pour installer PLY sur votre machine pour python2 / 3, procédez comme suit:

1. Téléchargez le code source à partir d' ici .
2. Décompressez le fichier zip téléchargé
3. Naviguez dans le dossier `ply-3.10` décompressé
4. Exécutez la commande suivante dans votre terminal: `python setup.py install`

Si vous avez terminé tout ce qui précède, vous devriez maintenant pouvoir utiliser le module PLY. Vous pouvez le tester en ouvrant un interpréteur python et en tapant `import ply.lex` .

Remarque: N'utilisez *pas* `pip` pour installer PLY, il installera une distribution défectueuse sur votre machine.

Le "Bonjour, Monde!" de PLY - Une calculatrice simple

Démontrons la puissance de PLY avec un exemple simple: ce programme prendra une expression arithmétique comme entrée de chaîne et tentera de la résoudre.

Ouvrez votre éditeur préféré et copiez le code suivant:

```
from ply import lex
import ply.yacc as yacc

tokens = (
    'PLUS',
    'MINUS',
    'TIMES',
    'DIV',
    'LPAREN',
```

```
    'RPAREN',
    'NUMBER',
)

t_ignore = ' \t'

t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIV     = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

def t_NUMBER( t ) :
    r'[0-9]+'
    t.value = int( t.value )
    return t

def t_newline( t ):
  r'\n+'
  t.lexer.lineno += len( t.value )

def t_error( t ):
  print("Invalid Token:",t.value[0])
  t.lexer.skip( 1 )

lexer = lex.lex()

precedence = (
    ( 'left', 'PLUS', 'MINUS' ),
    ( 'left', 'TIMES', 'DIV' ),
    ( 'nonassoc', 'UMINUS' )
)

def p_add( p ) :
    'expr : expr PLUS expr'
    p[0] = p[1] + p[3]

def p_sub( p ) :
    'expr : expr MINUS expr'
    p[0] = p[1] - p[3]

def p_expr2uminus( p ) :
    'expr : MINUS expr %prec UMINUS'
    p[0] = - p[2]

def p_mult_div( p ) :
    '''expr : expr TIMES expr
            | expr DIV expr'''

    if p[2] == '*' :
        p[0] = p[1] * p[3]
    else :
        if p[3] == 0 :
            print("Can't divide by 0")
            raise ZeroDivisionError('integer division by 0')
        p[0] = p[1] / p[3]

def p_expr2NUM( p ) :
    'expr : NUMBER'
    p[0] = p[1]
```

```
def p_parens( p ) :
    'expr : LPAREN expr RPAREN'
    p[0] = p[2]

def p_error( p ):
    print("Syntax error in input!")

parser = yacc.yacc()

res = parser.parse("-4*-(3-5)") # the input
print(res)
```

Enregistrez ce fichier sous le nom `calc.py` et exécutez-le.

Sortie:

```
-8
```

Quelle est la bonne réponse pour `-4 * - (3 - 5)` .

Partie 1: Tokenizing Input avec Lex

Il y a deux étapes que le code de l' exemple 1 effectués: l' un a été l'entrée de *jetons,* ce qui signifie qu'il a cherché symboles qui constituent l'expression arithmétique, et la seconde étape est l' *analyse,* qui consiste à analyser les jetons extraits et évaluer le résultat.

Cette section fournit un exemple simple de *tokenize de* saisie utilisateur, puis la décompose ligne par ligne.

```
    import ply.lex as lex

    # List of token names. This is always required
    tokens = [
       'NUMBER',
       'PLUS',
       'MINUS',
       'TIMES',
       'DIVIDE',
       'LPAREN',
       'RPAREN',
    ]

    # Regular expression rules for simple tokens
    t_PLUS    = r'\+'
    t_MINUS   = r'-'
    t_TIMES   = r'\*'
    t_DIVIDE  = r'/'
    t_LPAREN  = r'\('
    t_RPAREN  = r'\)'

    # A regular expression rule with some action code
    def t_NUMBER(t):
        r'\d+'
        t.value = int(t.value)
        return t
```

```
# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok)
```

Enregistrez ce fichier sous le nom `calclex.py` . Nous l'utiliserons lors de la construction de notre analyseur Yacc.

Panne

1. Importer le module en utilisant `import ply.lex`

2. Tous les lexers doivent fournir une liste appelée `tokens` qui définit tous les noms de jetons possibles pouvant être produits par le lexer. Cette liste est toujours requise.

   ```
   tokens = [
      'NUMBER',
      'PLUS',
      'MINUS',
      'TIMES',
      'DIVIDE',
      'LPAREN',
      'RPAREN',
   ]
   ```

`tokens` peuvent aussi être un tuple de chaînes (plutôt qu'une chaîne), où chaque chaîne indique un jeton comme auparavant.

3. La règle de regex pour chaque chaîne peut être définie comme une chaîne ou une fonction. Dans les deux cas, le nom de la variable doit être préfixé par t_ pour indiquer qu'il s'agit d'une règle pour les jetons correspondants.

- Pour les jetons simples, l'expression régulière peut être spécifiée sous la forme de chaînes: `t_PLUS = r'\+'`
- Si une action doit être exécutée, une règle de jeton peut être spécifiée en tant que fonction.

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Notez que la règle est spécifiée comme une chaîne de caractères dans la fonction. La fonction accepte un argument qui est une instance de `LexToken` , effectue une action puis renvoie l'argument.

Si vous souhaitez utiliser une chaîne externe comme règle d'expression régulière pour la fonction au lieu de spécifier une chaîne de document, prenez l'exemple suivant:

```
@TOKEN(identifier)          # identifier is a string holding the regex
def t_ID(t):
    ...      # actions
```

- Une instance d'objet `LexToken` (appelons cet objet `t`) a les attributs suivants:

1. `t.type` qui est le type de jeton (sous forme de chaîne) (par exemple: `'NUMBER'` , `'PLUS'` , etc.). Par défaut, `t.type` est défini sur le nom suivant le préfixe `t_` .
2. `t.value` qui est le lexème (le texte réel correspondant)
3. `t.lineno` qui est le numéro de ligne actuel (ce n'est pas automatiquement mis à jour, car le lexer ne sait rien des numéros de ligne). Mettez à jour lineno en utilisant une fonction appelée `t_newline` .

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

4. `t.lexpos` qui est la position du jeton par rapport au début du texte saisi.

- Si rien n'est renvoyé par une fonction de règle d'expression régulière, le jeton est ignoré. Si vous souhaitez ignorer un jeton, vous pouvez également ajouter le préfixe t_ignore_ à une variable de règle d'expression régulière au lieu de définir une fonction pour la même règle.

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # No return value. Token discarded
```

...Est le même que:

```
   t_ignore_COMMENT = r'\#.*'
```

Ceci est bien sûr invalide si vous effectuez une action lorsque vous voyez un commentaire. Dans ce cas, utilisez une fonction pour définir la règle regex.

Si vous n'avez pas défini de jeton pour certains caractères mais que vous souhaitez toujours l'ignorer, utilisez `t_ignore = "<characters to ignore>"` (ces préfixes sont nécessaires):

```
   t_ignore_COMMENT = r'\#.*'
   t_ignore  = ' \t'    # ignores spaces and tabs
```

- Lors de la création de l'expression rationnelle principale, lex ajoutera les expressions régulières spécifiées dans le fichier comme suit:

    1. Les jetons définis par les fonctions sont ajoutés dans le même ordre qu'ils apparaissent dans le fichier.
    2. Les jetons définis par des chaînes sont ajoutés par ordre décroissant de la longueur de chaîne de la chaîne définissant l'expression régulière pour ce jeton.

    Si vous faites correspondre `==` et `=` dans le même fichier, profitez de ces règles.

- Les littéraux sont des jetons renvoyés tels quels. `t.type` et `t.value` seront tous deux définis sur le caractère lui-même. Définir une liste de littéraux en tant que tels:

```
literals = [ '+', '-', '*', '/' ]
```

ou,

```
literals = "+-*/"
```

Il est possible d'écrire des fonctions de jeton qui effectuent des actions supplémentaires lorsque les littéraux sont mis en correspondance. Cependant, vous devrez définir le type de jeton de manière appropriée. Par exemple:

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = '{'  # Set token type to the expected literal (ABSOLUTE MUST if this
is a literal)
    return t
```

- Gérer les erreurs avec la fonction t_error.

```
# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1) # skip the illegal token (don't process it)
```

En général, `t.lexer.skip(n)` ignore n caractères dans la chaîne d'entrée.

4. Préparations finales:

   Construisez le lexer en utilisant `lexer = lex.lex()` .

   Vous pouvez également tout mettre dans une classe et appeler une instance use de la classe pour définir le lexer. Par exemple:

```
 import ply.lex as lex
 class MyLexer(object):
       ...     # everything relating to token rules and error handling comes here as
usual

       # Build the lexer
       def build(self, **kwargs):
           self.lexer = lex.lex(module=self, **kwargs)

       def test(self, data):
           self.lexer.input(data)
           for token in self.lexer.token():
               print(token)

       # Build the lexer and try it out

 m = MyLexer()
 m.build()           # Build the lexer
 m.test("3 + 4")     #
```

   Fournit une entrée à l'aide de `lexer.input(data)` où data est une chaîne

   Pour obtenir les jetons, utilisez `lexer.token()` qui renvoie les jetons correspondants. Vous pouvez itérer sur lexer en boucle comme dans:

```
for i in lexer:
    print(i)
```

Partie 2: Analyse d'entrées Tokenized avec Yacc

Cette section explique comment est traitée la saisie de jetons de la partie 1 - elle est effectuée à l'aide de grammaires sans contexte (CFG). La grammaire doit être spécifiée et les jetons sont traités en fonction de la grammaire. Sous le capot, l'analyseur utilise un analyseur LALR.

```
# Yacc example

import ply.yacc as yacc

# Get the token map from the lexer. This is required.
from calclex import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]
```

```
def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

while True:
   try:
       s = raw_input('calc > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
   print(result)
```

Panne

- Chaque règle de grammaire est définie par une fonction dans laquelle docstring à cette fonction contient la spécification de grammaire sans contexte appropriée. Les instructions qui constituent le corps de la fonction implémentent les actions sémantiques de la règle. Chaque fonction accepte un seul argument p qui est une séquence contenant les valeurs de chaque symbole de grammaire dans la règle correspondante. Les valeurs de `p[i]` sont mappées sur des symboles de grammaire, comme indiqué ici:

  ```
  def p_expression_plus(p):
      'expression : expression PLUS term'
  ```

```
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

- Pour les jetons, la "valeur" du `p[i]` est identique à l'attribut `p.value` attribué dans le module lexer. Donc, `PLUS` aura la valeur `+` .
- Pour les non-terminaux, la valeur est déterminée par tout ce qui est placé dans `p[0]` . Si rien n'est placé, la valeur est Aucun. De plus, `p[-1]` n'est pas la même chose que `p[3]` , puisque `p` n'est pas une simple liste (`p[-1]` peut spécifier des actions incorporées (non discuté ici)).

Notez que la fonction peut avoir n'importe quel nom, tant qu'elle est précédée de `p_` .

- La `p_error(p)` est définie pour intercepter les erreurs de syntaxe (comme `yyerror` dans yacc / bison).
- Plusieurs règles de grammaire peuvent être combinées en une seule fonction, ce qui est une bonne idée si les productions ont une structure similaire.

```
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
       term       : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

- Les littéraux de caractères peuvent être utilisés à la place des jetons.

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
       term       : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

  Bien entendu, les littéraux doivent être spécifiés dans le module lexer.

- Les productions vides ont la forme `'''symbol : '''`

- Pour définir explicitement le symbole de début, utilisez `start = 'foo'` , où `foo` est un non-terminal.
- La définition de la priorité et de l'associativité peut être effectuée à l'aide de la variable de priorité.

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'),  # Nonassociative operators
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),            # Unary minus operator
)
```

  Les jetons sont classés de la plus basse à la plus haute priorité. `nonassoc` signifie que ces jetons ne sont pas associés. Cela signifie que quelque chose comme `a < b < c` est illégal alors `a < b` est toujours légal.

- `parser.out` est un fichier de débogage créé lorsque le programme yacc est exécuté pour la première fois. Chaque fois qu'un changement / une réduction de conflit se produit, l'analyseur se déplace toujours.

Lire Python Lex-Yacc en ligne: https://riptutorial.com/fr/python/topic/10510/python-lex-yacc

Chapitre 166: Python Requests Post

Introduction

Documentation du module Requêtes Python dans le contexte de la méthode HTTP POST et de sa fonction Requests correspondante

Examples

Simple Post

```
from requests import post

foo = post('http://httpbin.org/post', data = {'key':'value'})
```

Réalisera une simple opération HTTP POST. Les données publiées peuvent être dans les formats les plus courants, mais les paires de valeurs clés sont les plus répandues.

Les entêtes

Les en-têtes peuvent être visualisés:

```
print(foo.headers)
```

Un exemple de réponse:

```
{'Content-Length': '439', 'X-Processed-Time': '0.000802993774414', 'X-Powered-By': 'Flask',
'Server': 'meinheld/0.6.1', 'Connection': 'keep-alive', 'Via': '1.1 vegur', 'Access-Control-
Allow-Credentials': 'true', 'Date': 'Sun, 21 May 2017 20:56:05 GMT', 'Access-Control-Allow-
Origin': '*', 'Content-Type': 'application/json'}
```

Les en-têtes peuvent également être préparés avant la publication:

```
headers = {'Cache-Control':'max-age=0',
        'Upgrade-Insecure-Requests':'1',
        'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/54.0.2840.99 Safari/537.36',
        'Content-Type':'application/x-www-form-urlencoded',
        'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
        'Referer':'https://www.groupon.com/signup',
        'Accept-Encoding':'gzip, deflate, br',
        'Accept-Language':'es-ES,es;q=0.8'
        }

 foo = post('http://httpbin.org/post', headers=headers, data = {'key':'value'})
```

Codage

Le codage peut être défini et visualisé de la même manière:

```
 print(foo.encoding)

'utf-8'

foo.encoding = 'ISO-8859-1'
```

Vérification SSL

Requests par défaut valide les certificats SSL des domaines. Cela peut être remplacé:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, verify=False)
```

Redirection

Toute redirection sera suivie (par exemple http à https), cela peut également être modifié:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, allow_redirects=False)
```

Si la post-opération a été redirigée, cette valeur est accessible:

```
print(foo.url)
```

Un historique complet des redirections peut être consulté:

```
print(foo.history)
```

Données codées par formulaire

```
from requests import post

payload = {'key1' : 'value1',
           'key2' : 'value2'
           }

foo = post('http://httpbin.org/post', data=payload)
```

Pour transmettre des données encodées avec la post-opération, les données doivent être structurées en tant que dictionnaire et fournies en tant que paramètre de données.

Si les données ne veulent pas être encodées, il suffit de transmettre une chaîne ou un entier au paramètre data.

Fournissez le dictionnaire au paramètre json pour que Requests formate automatiquement les données:

```
from requests import post

payload = {'key1' : 'value1', 'key2' : 'value2'}

foo = post('http://httpbin.org/post', json=payload)
```

Téléchargement de fichiers

Avec le module Requests, il suffit de fournir un `.read()` fichier par opposition au contenu récupéré avec `.read()` :

```
from requests import post

files = {'file' : open('data.txt', 'rb')}

foo = post('http://http.org/post', files=files)
```

Le nom de fichier, le type de contenu et les en-têtes peuvent également être définis:

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel',
{'Expires': '0'})}

foo = requests.post('http://httpbin.org/post', files=files)
```

Les chaînes peuvent également être envoyées sous forme de fichier, tant qu'elles sont fournies en tant que paramètre de `files` .

Plusieurs fichiers

Plusieurs fichiers peuvent être fournis à peu près comme un seul fichier:

```
multiple_files = [
    ('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),
    ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]

foo = post('http://httpbin.org/post', files=multiple_files)
```

Les réponses

Les codes de réponse peuvent être consultés à partir d'une opération postérieure:

```
from requests import post

foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.status_code)
```

Données renvoyées

Accéder aux données renvoyées:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.text)
```

Réponses brutes

Dans les cas où vous devez accéder à l’objet urllib3 response.HTTPResponse sous-jacent, vous pouvez procéder comme suit:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
res = foo.raw

print(res.read())
```

Authentification

Authentification HTTP simple

L'authentification HTTP simple peut être réalisée avec les éléments suivants:

```
from requests import post

foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

C'est techniquement court pour ce qui suit:

```
from requests import post
from requests.auth import HTTPBasicAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

Authentification HTTP Digest

L'authentification HTTP Digest se fait de manière très similaire, Requests fournit un objet différent pour cela:

```
from requests import post
from requests.auth import HTTPDigestAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0',
'natas0'))
```

Authentification personnalisée

Dans certains cas, les mécanismes d'authentification intégrés peuvent ne pas suffire, imaginez cet exemple:

Un serveur est configuré pour accepter l'authentification si l'expéditeur dispose de la chaîne d'agent utilisateur appropriée, d'une certaine valeur d'en-tête et fournit les informations d'identification correctes via l'authentification de base HTTP. Pour y parvenir, une classe d'authentification personnalisée doit être préparée, sous-classant AuthBase, qui est la base pour les implémentations d'authentification Requests:

```
from requests.auth import AuthBase
from requests.auth import _basic_auth_str
from requests._internal_utils import to_native_string

class CustomAuth(AuthBase):

    def __init__(self, secret_header, user_agent , username, password):
        # setup any auth-related data here
```

```
        self.secret_header =  secret_header
        self.user_agent = user_agent
        self.username = username
        self.password = password

    def __call__(self, r):
        # modify and return the request
        r.headers['X-Secret'] = self.secret_header
        r.headers['User-Agent'] = self.user_agent
        r.headers['Authorization'] = _basic_auth_str(self.username, self.password)

        return r
```

Cela peut ensuite être utilisé avec le code suivant:

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

Des procurations

Chaque opération POST de requête peut être configurée pour utiliser des proxy réseau

Proxy HTTP / S

```
from requests import post

proxies = {
  'http': 'http://192.168.0.128:3128',
  'https': 'http://192.168.0.127:1080',
   }

foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

L'authentification de base HTTP peut être fournie de cette manière:

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

SOCKS Proxies

L'utilisation de proxys de chaussettes nécessite des `requests[socks]` dépendances tierces `requests[socks]` , une fois installées, les proxys de chaussettes sont utilisés de manière très similaire à HTTPBasicAuth:

```
proxies = {
'http': 'socks5://user:pass@host:port',
'https': 'socks5://user:pass@host:port'
}

foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Lire Python Requests Post en ligne: https://riptutorial.com/fr/python/topic/10021/python-requests-post

Chapitre 167: Recherche

Remarques

Tous les algorithmes de recherche sur les itérables contenant `n` éléments ont une complexité `O(n)` . Seuls les algorithmes spécialisés comme `bisect.bisect_left()` peuvent être plus rapides avec la complexité `O(log(n))` .

Examples

Obtenir l'index des chaînes: str.index (), str.rindex () et str.find (), str.rfind ()

`String` également une méthode d' `index` mais aussi des options plus avancées et le `str.find` supplémentaire. Pour les deux, il existe une méthode *inversée* complémentaire.

```
astring = 'Hello on StackOverflow'
astring.index('o')  # 4
astring.rindex('o') # 20

astring.find('o')   # 4
astring.rfind('o')  # 20
```

La différence entre `index` / `rindex` et `find` / `rfind` est ce qui arrive si la sous-chaîne n'est pas trouvée dans la chaîne:

```
astring.index('q') # ValueError: substring not found
astring.find('q')  # -1
```

Toutes ces méthodes permettent un index de début et de fin:

```
astring.index('o', 5)    # 6
astring.index('o', 6)    # 6 - start is inclusive
astring.index('o', 5, 7) # 6
astring.index('o', 5, 6) #  - end is not inclusive
```

ValueError: sous-chaîne introuvable

```
astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - still from left to right

astring.rindex('o', 4, 7) # 6
```

Recherche d'un élément

Toutes les collections intégrées dans Python implémentent un moyen de vérifier l'appartenance à un élément en utilisant `in` .

liste

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist   # True
10 in alist  # False
```

Tuple

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple    # False
'4' in atuple  # True
```

Chaîne

```
astring = 'i am a string'
'a' in astring   # True
'am' in astring  # True
'I' in astring   # False
```

Ensemble

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset  # True
10 in aset        # False
```

Dict

`dict` est un peu spécial: la normale `in` vérifie que les *clés.* Si vous souhaitez rechercher des *valeurs,* vous devez le spécifier. La même chose si vous souhaitez rechercher *des* paires *clé-valeur* .

```
adict = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
1 in adict                 # True   - implicitly searches in keys
'a' in adict               # False
2 in adict.keys()          # True   - explicitly searches in keys
'a' in adict.values()      # True   - explicitly searches in values
(0, 'a') in adict.items()  # True   - explicitly searches key/value pairs
```

Obtenir la liste d'index et les tuples: list.index (), tuple.index ()

`list` et `tuple` ont une méthode d' `index` pour obtenir la position de l'élément:

```
alist = [10, 16, 26, 5, 2, 19, 105, 26]
# search for 16 in the list
alist.index(16) # 1
alist[1]        # 16

alist.index(15)
```

ValueError: 15 n'est pas dans la liste

Mais ne retourne que la position du premier élément trouvé:

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26)   # 2
atuple[2]          # 26
atuple[7]          # 26 - is also 26!
```

Recherche de clé (s) pour une valeur dans dict

`dict` n'ont pas de méthode intégrée pour rechercher une valeur ou une clé car les *dictionnaires ne* sont pas ordonnés. Vous pouvez créer une fonction qui obtient la clé (ou les clés) pour une valeur spécifiée:

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[key] == value:
            foundkeys.append(key)
    return foundkeys
```

Cela pourrait également être écrit comme une liste de compréhension équivalente:

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

Si vous ne vous souciez que d'une clé trouvée:

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

Les deux premières fonctions renverront une `list` de toutes les `keys` ayant la valeur spécifiée:

```
adict = {'a': 10, 'b': 20, 'c': 10}
getKeysForValue(adict, 10)     # ['c', 'a'] - order is random could as well be ['a', 'c']
getKeysForValueComp(adict, 10) # ['c', 'a'] - dito
getKeysForValueComp(adict, 20) # ['b']
getKeysForValueComp(adict, 25) # []
```

L'autre ne renverra qu'une clé:

```
getOneKeyForValue(adict, 10)   # 'c'  - depending on the circumstances this could also be 'a'
getOneKeyForValue(adict, 20)   # 'b'
```

et `StopIteration` une `StopIteration` - `Exception` si la valeur n'est pas dans le `dict` :

```
getOneKeyForValue(adict, 25)
```

StopIteration

Obtenir l'index des séquences triées: bisect.bisect_left ()

Les séquences triées permettent l'utilisation d'algorithmes de recherche plus rapides: `bisect.bisect_left()` [1] :

```
import bisect

def index_sorted(sorted_seq, value):
    """Locate the leftmost value exactly equal to x or raise a ValueError"""
    i = bisect.bisect_left(sorted_seq, value)
    if i != len(sorted_seq) and sorted_seq[i] == value:
        return i
    raise ValueError

alist = [i for i in range(1, 100000, 3)] # Sorted list from 1 to 100000 with step 3
index_sorted(alist, 97285) # 32428
index_sorted(alist, 4)     # 1
index_sorted(alist, 97286)
```

ValeurErreur

Pour les très grandes **séquences triées,** le gain de vitesse peut être assez élevé. Pour la première recherche, environ 500 fois plus rapidement:

```
%timeit index_sorted(alist, 97285)
# 100000 loops, best of 3: 3 µs per loop
%timeit alist.index(97285)
# 1000 loops, best of 3: 1.58 ms per loop
```

Bien que ce soit un peu plus lent si l'élément est l'un des tous premiers:

```
%timeit index_sorted(alist, 4)
# 100000 loops, best of 3: 2.98 µs per loop
%timeit alist.index(4)
# 1000000 loops, best of 3: 580 ns per loop
```

Recherche de séquences imbriquées

La recherche dans des séquences imbriquées comme une `list` de `tuple` nécessite une approche telle que la recherche de clés dans les `dict` mais nécessite des fonctions personnalisées.

L'index de la séquence la plus externe si la valeur a été trouvée dans la séquence:

```
def outer_index(nested_sequence, value):
    return next(index for index, inner in enumerate(nested_sequence)
                      for item in inner
                      if item == value)

alist_of_tuples = [(4, 5, 6), (3, 1, 'a'), (7, 0, 4.3)]
outer_index(alist_of_tuples, 'a')  # 1
outer_index(alist_of_tuples, 4.3)  # 2
```

ou l'index de la séquence externe et interne:

```
def outer_inner_index(nested_sequence, value):
    return next((oindex, iindex) for oindex, inner in enumerate(nested_sequence)
                                 for iindex, item in enumerate(inner)
                                 if item == value)

outer_inner_index(alist_of_tuples, 'a') # (1, 2)
alist_of_tuples[1][2]  # 'a'

outer_inner_index(alist_of_tuples, 7)   # (2, 0)
alist_of_tuples[2][0]  # 7
```

En général (*pas toujours*) utiliser `next` et une **expression de générateur** avec des conditions pour trouver la première occurrence de la valeur recherchée est l'approche la plus efficace.

Recherche dans des classes personnalisées: __contains__ et __iter__

Pour autoriser l'utilisation de `in` pour les classes personnalisées, la classe doit fournir la méthode magique `__contains__` ou, à défaut, une `__iter__` __iter__.

Supposons que vous ayez une classe contenant une `list` de `list` s:

```
class ListList:
    def __init__(self, value):
        self.value = value
        # Create a set of all values for fast access
        self.setofvalues = set(item for sublist in self.value for item in sublist)

    def __iter__(self):
        print('Using __iter__.')
        # A generator over all sublist elements
        return (item for sublist in self.value for item in sublist)

    def __contains__(self, value):
        print('Using __contains__.')
        # Just lookup if the value is in the set
        return value in self.setofvalues

        # Even without the set you could use the iter method for the contains-check:
        # return any(item == value for item in iter(self))
```

L'utilisation du test d'appartenance est possible en utilisant `in` :

```
a = ListList([[1,1,1],[0,1,1],[1,5,1]])
10 in a    # False
# Prints: Using __contains__.
5 in a     # True
# Prints: Using __contains__.
```

même après la suppression de la méthode `__contains__` :

```
del ListList.__contains__
5 in a     # True
# Prints: Using __iter__.
```

Remarque: La boucle `in` (comme `for i in a`) utilisera toujours `__iter__` même si la classe implémente une `__contains__` méthode.

Lire Recherche en ligne: https://riptutorial.com/fr/python/topic/350/recherche

Chapitre 168: Reconnaissance optique de caractères

Introduction

La reconnaissance optique de caractères consiste à convertir des images de texte en texte réel. Dans ces exemples, trouvez des moyens d'utiliser l'OCR en python.

Examples

PyTesseract

PyTesseract est un package Python en développement pour OCR.

Utiliser PyTesseract est assez simple:

```
try:
    import Image
except ImportError:
    from PIL import Image

import pytesseract

    #Basic OCR
    print(pytesseract.image_to_string(Image.open('test.png')))

    #In French
    print(pytesseract.image_to_string(Image.open('test-european.jpg'), lang='fra'))
```

PyTesseract est open source et peut être trouvé ici .

PyOCR

`PyOCR` est un autre module dont le code source est ici .

Aussi simple à utiliser et a plus de fonctionnalités que `PyTesseract` .

Pour initialiser:

```
from PIL import Image
import sys

import pyocr
import pyocr.builders

tools = pyocr.get_available_tools()
# The tools are returned in the recommended order of usage
tool = tools[0]
```

```
langs = tool.get_available_languages()
lang = langs[0]
# Note that languages are NOT sorted in any way. Please refer
# to the system locale settings for the default language
# to use.
```

Et quelques exemples d'utilisation:

```
txt = tool.image_to_string(
    Image.open('test.png'),
    lang=lang,
    builder=pyocr.builders.TextBuilder()
)
# txt is a Python string

word_boxes = tool.image_to_string(
    Image.open('test.png'),
    lang="eng",
    builder=pyocr.builders.WordBoxBuilder()
)
# list of box objects. For each box object:
#   box.content is the word in the box
#   box.position is its position on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

line_and_word_boxes = tool.image_to_string(
    Image.open('test.png'), lang="fra",
    builder=pyocr.builders.LineBoxBuilder()
)
# list of line objects. For each line object:
#   line.word_boxes is a list of word boxes (the individual words in the line)
#   line.content is the whole text of the line
#   line.position is the position of the whole line on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

# Digits - Only Tesseract (not 'libtesseract' yet !)
digits = tool.image_to_string(
    Image.open('test-digits.png'),
    lang=lang,
    builder=pyocr.tesseract.DigitBuilder()
)
# digits is a python string
```

Lire Reconnaissance optique de caractères en ligne:
https://riptutorial.com/fr/python/topic/9302/reconnaissance-optique-de-caracteres

Chapitre 169: Récursivité

Remarques

La récursivité nécessite une condition d'arrêt `stopCondition` pour sortir de la récursivité.

La variable d'origine doit être transmise à la fonction récursive pour être stockée.

Examples

Somme des nombres de 1 à n

Si je voulais trouver la somme des nombres de `1` à `n` où `n` est un nombre naturel, je peux faire `1 + 2 + 3 + 4 + ... + (several hours later) + n`. Sinon, je pourrais écrire une boucle `for` :

```
n = 0
for i in range (1, n+1):
    n += i
```

Ou je pourrais utiliser une technique connue sous le nom de récursivité:

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

La récursivité présente des avantages par rapport aux deux méthodes ci-dessus. La récursion prend moins de temps que l'écriture de `1 + 2 + 3` pour une somme de 1 à 3. Pour la `recursion(4)` , la récursion peut être utilisée pour reculer:

Appels de fonction: (4 -> 4 + 3 -> 4 + 3 + 2 -> 4 + 3 + 2 + 1 -> 10)

Alors que la boucle `for` fonctionne strictement en avant: (1 -> 1 + 2 -> 1 + 2 + 3 -> 1 + 2 + 3 + 4 -> 10). Parfois, la solution récursive est plus simple que la solution itérative. Cela est évident lors de la mise en œuvre d'une inversion d'une liste chaînée.

Le quoi, comment et quand de récursivité

La récursivité se produit lorsqu'un appel de fonction provoque l'appel de la même fonction avant la fin de l'appel de fonction d'origine. Par exemple, considérons l'expression mathématique bien connue `x!` (c'est-à-dire l'opération factorielle). L'opération factorielle est définie pour tous les entiers non négatifs comme suit:

- Si le nombre est 0, la réponse est 1.
- Sinon, la réponse est ce nombre multiplié par la factorielle de moins d'un chiffre.

En Python, une implémentation naïve de l'opération factorielle peut être définie comme une

fonction comme suit:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Les fonctions de récursion peuvent être difficiles à saisir parfois, alors parcourons cette étape par étape. Considérons l'expression `factorial(3)` . Ceci et *tous* les appels de fonctions créent un nouvel **environnement** . Un environnement est simplement une table qui mappe les identifiants (par exemple, `n` , `factorial` , `print` , etc.) à leurs valeurs correspondantes. À tout moment, vous pouvez accéder à l'environnement actuel en utilisant les utilisateurs `locals()` . Dans le premier appel de fonction, la seule variable locale définie est `n = 3` . Par conséquent, imprimer les `locals()` afficherait `{'n': 3}` . Puisque `n == 3` , la valeur de retour devient `n * factorial(n - 1)` .

À cette étape suivante, les choses peuvent devenir un peu confuses. En regardant notre nouvelle expression, nous savons déjà ce que `n` est. Cependant, nous ne savons pas encore quelle `factorial(n - 1)` est. Tout d'abord, `n - 1` évalué à `2` . Ensuite, `2` est passé à `factorial` comme valeur pour `n` . Puisqu'il s'agit d'un nouvel appel de fonction, un deuxième environnement est créé pour stocker ce nouveau `n` . Soit *A* le premier environnement et *B* le deuxième environnement. *Un* existe toujours et est égal à `{'n': 3}` , cependant, *B* (qui est égal à `{'n': 2}`) est l'environnement actuel. En regardant le corps de la fonction, la valeur de retour est, encore une fois, `n * factorial(n - 1)` . Sans évaluer cette expression, substituons-la à l'expression de retour d'origine. En faisant cela, nous mentalement rejeter *B,* alors pensez à remplacer `n` en conséquence (ie les références à *B* « s `n` sont remplacés par `n - 1` qui utilise *un* » s `n`). Maintenant, l'expression de retour d'origine devient `n * ((n - 1) * factorial((n - 1) - 1))` . Prenez une seconde pour vous assurer que vous comprenez pourquoi.

Maintenant, évaluons la partie `factorial((n - 1) - 1))` de celle-ci. Puisque *A* est `n == 3` , nous passons `1` en `factorial` . Par conséquent, nous créons un nouvel environnement *C* qui est égal à `{'n': 1}` . Là encore, la valeur de retour est `n * factorial(n - 1)` . Alors, remplaçons `factorial((n - 1) - 1))` de l'expression de retour «original» de la même manière que nous avons ajusté l'expression de retour d'origine plus tôt. L'expression «originale» est maintenant `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))` .

Presque fini. Maintenant, nous devons évaluer `factorial((n - 2) - 1)` . Cette fois, nous passons à `0` . Par conséquent, cela évalue à `1` . Maintenant, effectuons notre dernière substitution. L'expression de retour «originale» est maintenant `n * ((n - 1) * ((n - 2) * 1))` . Rappelant que l'expression de retour d'origine est évaluée sous *A* , l'expression devient `3 * ((3 - 1) * ((3 - 2) * 1))` . Cela, bien sûr, évalue à 6. Pour confirmer que c'est la bonne réponse, rappelez-vous que `3! == 3 * 2 * 1 == 6` . Avant de poursuivre la lecture, assurez-vous de bien comprendre le concept d'environnement et son application à la récursivité.

L'instruction `if n == 0: return 1` s'appelle un cas de base. En effet, il ne présente aucune récursivité. Un cas de base est absolument nécessaire. Sans celui-ci, vous rencontrerez une récursion infinie. Cela dit, tant que vous avez au moins un cas de base, vous pouvez avoir autant de cas que vous le souhaitez. Par exemple, nous pourrions avoir une `factorial` écrite de la

manière suivante:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Vous pouvez également avoir plusieurs cas de récursivité, mais nous n'entrerons pas dans cette perspective, car elle est relativement rare et souvent difficile à traiter mentalement.

Vous pouvez également avoir des appels de fonction récursifs «parallèles». Par exemple, considérons la séquence de Fibonacci qui est définie comme suit:

- Si le nombre est 0, la réponse est 0.
- Si le nombre est 1, la réponse est 1.
- Sinon, la réponse est la somme des deux nombres précédents de Fibonacci.

Nous pouvons définir cela comme suit:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

Je ne vais pas parcourir cette fonction aussi complètement que je l'ai fait avec `factorial(3)` , mais la valeur de retour finale de `fib(5)` est équivalente à l'expression suivante (*syntaxiquement* invalide):

```
(
  fib((n - 2) - 2)
  +
  (
    fib(((n - 2) - 1) - 2)
    +
    fib(((n - 2) - 1) - 1)
  )
)
+
(
  (
    fib(((n - 1) - 2) - 2)
    +
    fib(((n - 1) - 2) - 1)
  )
  +
  (
    fib(((n - 1) - 1) - 2)
    +
    (
      fib((((n - 1) - 1) - 1) - 2)
      +
```

```
      fib((((n - 1) - 1) - 1) - 1)
    )
  )
)
```

Cela devient `(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))` qui bien sûr est évalué à `5` .

Passons maintenant à quelques termes de vocabulaire supplémentaires:

- Un **appel de fin** est simplement un appel de fonction récursif qui est la dernière opération à effectuer avant de renvoyer une valeur. Pour être clair, `return foo(n - 1)` est un appel de fin, mais `return foo(n - 1) + 1` ne l'est pas (puisque l'ajout est la dernière opération).
- **L'optimisation des appels de queue** (TCO) est un moyen de réduire automatiquement la récursivité dans les fonctions récursives.
- **L'élimination des appels de queue** (TCE) est la réduction d'un appel de fin à une expression qui peut être évaluée sans récurrence. Le TCE est un type de TCO.

L'optimisation des appels de queue est utile pour plusieurs raisons:

- L'interpréteur peut minimiser la quantité de mémoire occupée par les environnements. Comme aucun ordinateur ne dispose d'une mémoire illimitée, des appels de fonction récursifs excessifs entraîneraient un **débordement de pile** .
- L'interpréteur peut réduire le nombre de commutateurs de **trames** de **pile** .

Python n'a aucune forme de TCO implémentée pour plusieurs raisons . Par conséquent, d'autres techniques sont nécessaires pour contourner cette limitation. La méthode de choix dépend du cas d'utilisation. Avec une certaine intuition, les définitions de `factorial` et de `fib` peuvent être facilement converties en code itératif comme suit:

```
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product

def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

C'est généralement le moyen le plus efficace d'éliminer manuellement la récursivité, mais cela peut devenir assez difficile pour des fonctions plus complexes.

Un autre outil utile est le décorateur lru_cache de Python, qui permet de réduire le nombre de calculs redondants.

Vous avez maintenant une idée de comment éviter la récursivité en Python, mais quand *devriez-*vous utiliser la récursivité? La réponse est «pas souvent». Toutes les fonctions récursives peuvent être implémentées de manière itérative. Il suffit simplement de trouver comment le faire.

Cependant, il existe de rares cas de récursivité. La récursivité est courante en Python lorsque les entrées attendues ne provoqueraient pas un nombre significatif d'appels de fonction récursifs.

Si la récursivité est un sujet qui vous intéresse, je vous implore d'étudier des langages fonctionnels tels que Scheme ou Haskell. Dans ces langues, la récursivité est beaucoup plus utile.

Veuillez noter que l'exemple ci-dessus pour la séquence Fibonacci, bien qu'il montre comment appliquer la définition en python et l'utilisation ultérieure du cache lru, a un temps d'exécution inefficace car il effectue 2 appels récursifs pour chaque cas non-base. Le nombre d'appels à la fonction augmente exponentiellement à `n` .
Plutôt non intuitivement, une implémentation plus efficace utiliserait la récursivité linéaire:

```
def fib(n):
    if n <= 1:
        return (n,0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

Mais celui-là a le problème de retourner une *paire* de chiffres. Cela souligne que certaines fonctions ne gagnent pas beaucoup de récursivité.

Exploration d'arbres avec récursion

Disons que nous avons l'arbre suivant:

```
root
- A
  - AA
  - AB
- B
  - BA
  - BB
    - BBA
```

Maintenant, si nous souhaitons énumérer tous les noms des éléments, nous pourrions le faire avec un simple for-loop. Nous supposons qu'il y a une fonction `get_name()` pour retourner une chaîne du nom d'un noeud, une fonction `get_children()` pour renvoyer une liste de tous les sous-noeuds d'un noeud donné dans l'arborescence et une fonction `get_root()` pour obtenir le noeud racine.

```
root = get_root(tree)
for node in get_children(root):
    print(get_name(node))
    for child in get_children(node):
        print(get_name(child))
        for grand_child in get_children(child):
            print(get_name(grand_child))
# prints: A, AA, AB, B, BA, BB, BBA
```

Cela fonctionne bien et rapidement, mais que se passe-t-il si les sous-nœuds ont leurs propres sous-nœuds? Et ces sous-noeuds pourraient avoir plus de sous-noeuds ... Et si vous ne savez

pas à l'avance combien il y en aura? Une méthode pour résoudre ce problème est l'utilisation de la récursivité.

```
def list_tree_names(node):
    for child in get_children(node):
        print(get_name(child))
        list_tree_names(node=child)

list_tree_names(node=get_root(tree))
# prints: A, AA, AB, B, BA, BB, BBA
```

Vous souhaitez peut-être ne pas imprimer, mais renvoyer une liste plate de tous les noms de nœuds. Cela peut être fait en passant une liste déroulante en paramètre.

```
def list_tree_names(node, lst=[]):
    for child in get_children(node):
        lst.append(get_name(child))
        list_tree_names(node=child, lst=lst)
    return lst

list_tree_names(node=get_root(tree))
# returns ['A', 'AA', 'AB', 'B', 'BA', 'BB', 'BBA']
```

Augmenter la profondeur de récursivité maximale

Il y a une limite à la profondeur de la récursion possible, qui dépend de l'implémentation de Python. Lorsque la limite est atteinte, une exception RuntimeError est déclenchée:

```
RuntimeError: Maximum Recursion Depth Exceeded
```

Voici un exemple de programme qui provoquerait cette erreur:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))
cursing(0)
# Out: I recursed 1083 times!
```

Il est possible de modifier la limite de profondeur de récursion en utilisant

```
sys.setrecursionlimit(limit)
```

Vous pouvez vérifier quels sont les paramètres actuels de la limite en exécutant:

```
sys.getrecursionlimit()
```

Utiliser la même méthode ci-dessus avec notre nouvelle limite que nous obtenons

```
sys.setrecursionlimit(2000)
```

```
cursing(0)
# Out: I recursed 1997 times!
```

À partir de Python 3.5, l'exception est une erreur Récursion, dérivée de RuntimeError.

Récursion de la queue - Mauvaise pratique

Lorsque la seule chose renvoyée par une fonction est un appel récursif, on parle de récursion de queue.

Voici un exemple de compte à rebours écrit en utilisant la récursion de queue:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Tout calcul pouvant être effectué à l'aide d'une itération peut également être effectué en utilisant la récursivité. Voici une version de find_max écrite en utilisant la récursion de la queue:

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

La récursion de la queue est considérée comme une mauvaise pratique en Python, car le compilateur Python ne gère pas l'optimisation des appels récursifs de queue. La solution récursive dans de tels cas utilise plus de ressources système que la solution itérative équivalente.

Optimisation de la récursion de la queue grâce à l'introspection de la pile

Par défaut, la pile de récurrence de Python ne peut pas dépasser 1000 images. Cela peut être changé en définissant le `sys.setrecursionlimit(15000)` qui est plus rapide cependant, cette méthode consomme plus de mémoire. Au lieu de cela, nous pouvons également résoudre le problème de la récursion de la queue en utilisant l'introspection de la pile.

```
#!/usr/bin/env python2.4
# This program shows off a python decorator which implements tail call optimization. It
# does this by throwing an exception if it is it's own grandparent, and catching such
# exceptions to recall the stack.

import sys

class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs
```

```
def tail_call_optimized(g):
"""
This function decorates a function with tail call
optimization. It does this by throwing an exception
if it is it's own grandparent, and catching such
exceptions to fake the tail call optimization.

This function fails if the decorated
function recurses in a non-tail context.
"""

    def func(*args, **kwargs):
        f = sys._getframe()
        if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
            raise TailRecurseException(args, kwargs)
        else:
            while 1:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException, e:
                    args = e.args
                    kwargs = e.kwargs
    func.__doc__ = g.__doc__
    return func
```

Pour optimiser les fonctions récursives, nous pouvons utiliser le décorateur `@tail_call_optimized` pour appeler notre fonction. Voici quelques exemples de récurrence courants utilisant le décorateur décrit ci-dessus:

Exemple factoriel:

```
@tail_call_optimized
def factorial(n, acc=1):
  "calculate a factorial"
  if n == 0:
    return acc
  return factorial(n-1, n*acc)

print factorial(10000)
# prints a big, big number,
# but doesn't hit the recursion limit.
```

Exemple Fibonacci:

```
@tail_call_optimized
def fib(i, current = 0, next = 1):
  if i == 0:
    return current
  else:
    return fib(i - 1, next, current + next)

print fib(10000)
# also prints a big number,
# but doesn't hit the recursion limit.
```

Lire Récursivité en ligne: https://riptutorial.com/fr/python/topic/1716/recursivite

Chapitre 170: Réduire

Syntaxe

- réduire (fonction, itérable [, initialiseur])

Paramètres

Paramètre	Détails
fonction	fonction utilisée pour réduire l'itérable (doit prendre deux arguments). (*uniquement positionnel*)
itérable	iterable qui va etre reduit. (*uniquement positionnel*)
initialiseur	valeur de départ de la réduction. (*facultatif* , *uniquement positionnel*)

Remarques

`reduce` peut ne pas toujours être la fonction la plus efficace. Pour certains types, il existe des fonctions ou des méthodes équivalentes:

- `sum()` pour la somme d'une séquence contenant des éléments à *ajouter* (pas des chaînes):

```
sum([1,2,3])                                    # = 6
```

- `str.join` pour la concaténation de chaînes:

```
''.join(['Hello', ',', ' World'])               # = 'Hello, World'
```

- `next` avec un générateur pourrait être une variante de court-circuit par rapport à `reduce` :

```
# First falsy item:
next((i for i in [100, [], 20, 0] if not i)) # = []
```

Examples

Vue d'ensemble

```
# No import needed


# No import required...
from functools import reduce # ... but it can be loaded from the functools module
```

```
from functools import reduce # mandatory
```

`reduce` réduit une itération en appliquant une fonction à plusieurs reprises sur l'élément suivant d'un résultat `iterable` et cumulatif jusqu'à présent.

```
def add(s1, s2):
    return s1 + s2

asequence = [1, 2, 3]

reduce(add, asequence)  # equivalent to: add(add(1,2),3)
# Out: 6
```

Dans cet exemple, nous avons défini notre propre fonction `add` . Cependant, Python est livré avec une fonction équivalente standard dans le module `operator` :

```
import operator
reduce(operator.add, asequence)
# Out: 6
```

`reduce` peut aussi être passé une valeur de départ:

```
reduce(add, asequence, 10)
# Out: 16
```

En utilisant réduire

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                           arg2=s2,
                                           res=s1*s2))
    return s1 * s2

asequence = [1, 2, 3]
```

Étant donné un `initializer` la fonction est démarrée en l'appliquant à l'initialiseur et au premier élément pouvant être itéré:

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30
```

Sans le paramètre `initializer` , la `reduce` commence en appliquant la fonction aux deux premiers éléments de la liste:

```
cumprod = reduce(multiply, asequence)
```

```
# Out: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
# Out: 6
```

Produit cumulatif

```
import operator
reduce(operator.mul, [10, 5, -3])
# Out: -150
```

Variante sans court-circuit de tout / tout

`reduce` ne sera pas fin à l'itération avant la `iterable` a été complètement itéré il peut être utilisé pour créer un court-circuit non `any()` ou `all()` fonction:

```
import operator
# non short-circuit "all"
reduce(operator.and_, [False, True, True, True]) # = False

# non short-circuit "any"
reduce(operator.or_, [True, False, False, False]) # = True
```

Premier élément de vérité / falsification d'une séquence (ou dernier élément s'il n'y en a pas)

```
# First falsy element or last element if all are truthy:
reduce(lambda i, j: i and j, [100, [], 20, 10])    # = []
reduce(lambda i, j: i and j, [100, 50, 20, 10])    # = 10

# First truthy element or last element if all falsy:
reduce(lambda i, j: i or j, [100, [], 20, 0])     # = 100
reduce(lambda i, j: i or j, ['', {}, [], None])   # = None
```

Au lieu de créer une fonction `lambda` il est généralement recommandé de créer une fonction nommée:

```
def do_or(i, j):
    return i or j

def do_and(i, j):
    return i and j

reduce(do_or, [100, [], 20, 0])                   # = 100
reduce(do_and, [100, [], 20, 0])                  # = []
```

Lire Réduire en ligne: https://riptutorial.com/fr/python/topic/328/reduire

Chapitre 171: Représentations de chaîne des instances de classe: méthodes __str__ et __repr__

Remarques

Une note sur l'implémentation des deux méthodes

Lorsque les deux méthodes sont implémentées, il est assez courant d'avoir une méthode `__str__` qui retourne une représentation conviviale (par exemple, "Ace of Spaces") et `__repr__` renvoie une représentation `eval` friendlyly.

En fait, les documents Python pour `repr()` notent exactement ceci:

> Pour de nombreux types, cette fonction tente de renvoyer une chaîne qui donnerait un objet ayant la même valeur lorsqu'elle est transmise à eval (), sinon la représentation est une chaîne entre crochets contenant le nom du type de l'objet. avec des informations supplémentaires comprenant souvent le nom et l'adresse de l'objet.

Cela signifie que `__str__` pourrait être implémenté pour renvoyer quelque chose comme "As of Spaces" comme indiqué précédemment, `__repr__` pourrait être implémenté pour renvoyer à la place `Card('Spades', 1)`

Cette chaîne pourrait être renvoyée directement dans `eval` en un peu comme un "aller-retour":

```
object -> string -> object
```

Un exemple d'implémentation d'une telle méthode pourrait être:

```
def __repr__(self):
    return "Card(%s, %d)" % (self.suit, self.pips)
```

Remarques

[1] Cette sortie est spécifique à l'implémentation. La chaîne affichée provient de cpython.

[2] Vous avez peut-être déjà vu le résultat de cette division `str()` / `repr()` sans le savoir. Lorsque des chaînes contenant des caractères spéciaux, tels que des barres obliques inverses, sont converties en chaînes via `str()` les barres obliques inverses apparaissent telles quelles (elles

apparaissent une fois). Lorsqu'elles sont converties en chaînes via `repr()` (par exemple, lorsque des éléments d'une liste sont affichés), les barres obliques inversées sont échappées et apparaissent donc deux fois.

Examples

Motivation

Vous venez donc de créer votre première classe en Python, une petite classe intéressante qui encapsule une carte à jouer:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips
```

Ailleurs dans votre code, vous créez quelques instances de cette classe:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)
```

Vous avez même créé une liste de cartes afin de représenter une "main":

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

Maintenant, pendant le débogage, vous voulez voir à quoi ressemble votre main, alors vous faites ce qui vient naturellement et écrivez:

```
print(my_hand)
```

Mais ce que vous obtenez est un tas de charabia:

```
[<__main__.Card instance at 0x0000000002533788>,
 <__main__.Card instance at 0x00000000025B95C8>,
 <__main__.Card instance at 0x00000000025FF508>]
```

Confus, vous essayez d'imprimer une seule carte:

```
print(ace_of_spades)
```

Et encore une fois, vous obtenez cette sortie bizarre:

```
<__main__.Card instance at 0x0000000002533788>
```

N'ai pas peur. Nous sommes sur le point de résoudre ce problème.

Tout d'abord, il est important de comprendre ce qui se passe ici. Lorsque vous avez écrit

`print(ace_of_spades)` vous avez dit à Python que vous souhaitiez imprimer des informations sur l'instance `Card` votre code appelle `ace_of_spades` . Et pour être juste, ça l'a fait.

Cette sortie est composée de deux bits importants: le `type` de l'objet et son `id` . La deuxième partie seule (le nombre hexadécimal) est suffisante pour identifier de manière unique l'objet au moment de l'appel d' `print` . [1]

Ce qui s'est vraiment passé, c'est que vous avez demandé à Python de "mettre en mots" l'essence de cet objet et de vous l'afficher ensuite. Une version plus explicite de la même machine pourrait être:

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```

Dans la première ligne, vous essayez de transformer votre instance de `Card` en chaîne et, dans le second, vous l'affichez.

Le problème

Le problème que vous rencontrez est dû au fait que, même si vous avez dit à Python tout ce qu'il fallait savoir sur la classe `Card` pour *créer des* cartes, vous *ne lui* avez *pas* indiqué comment convertir les instances `Card` en chaînes.

Et comme il ne savait pas, lorsque vous (implicitement) avez écrit `str(ace_of_spades)` , cela vous a donné ce que vous avez vu, une représentation générique de l'instance de la `Card` .

La solution (partie 1)

Mais *nous pouvons* dire à Python comment nous voulons que les instances de nos classes personnalisées soient converties en chaînes. Et la façon dont nous le faisons est avec la `__str__` "dunder" (pour double-underscore) ou "magic".

Chaque fois que vous demandez à Python de créer une chaîne à partir d'une instance de classe, celle-ci recherche une méthode `__str__` sur la classe et l'appelle.

Considérez la version mise à jour suivante de notre classe de `Card` :

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

        card_name = special_names.get(self.pips, str(self.pips))

        return "%s of %s" % (card_name, self.suit)
```

Ici, nous avons maintenant défini la méthode `__str__` sur notre classe `Card` qui, après une simple recherche dans les dictionnaires, **retourne** une chaîne formatée selon notre `__str__` .

(Notez que "return" est en gras ici, pour souligner l'importance de retourner une chaîne, et non simplement l'imprimer. L'impression peut sembler fonctionner, mais la carte est imprimée lorsque vous faites quelque chose comme `str(ace_of_spades)` , sans même avoir un appel de fonction d'impression dans votre programme principal. Pour être clair, assurez-vous que `__str__` renvoie une chaîne.).

La méthode `__str__` est une méthode, donc le premier argument sera `self` et il ne devrait ni accepter, ni transmettre des arguments supplémentaires.

Revenons à notre problème d'affichage de la carte d'une manière plus conviviale, si nous courons à nouveau:

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

Nous verrons que notre production est bien meilleure:

```
Ace of Spades
```

Tellement génial, nous avons fini, non?

Eh bien, juste pour couvrir nos bases, vérifions que nous avons résolu le premier problème rencontré, en imprimant la liste des instances de la `Card` , la `hand` .

Nous vérifions donc le code suivant:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

Et, à notre grande surprise, nous obtenons à nouveau ces codes hexadécimaux amusants:

```
[<__main__.Card instance at 0x00000000026F95C8>,
 <__main__.Card instance at 0x000000000273F4C8>,
 <__main__.Card instance at 0x0000000002732E08>]
```

Que se passe-t-il? Nous avons dit à Python comment nous voulions que nos instances de `Card` soient affichées, pourquoi semble-t-il apparemment oublier?

La solution (partie 2)

Eh bien, la machine en arrière-plan est un peu différente lorsque Python veut obtenir la représentation sous forme de chaîne des éléments dans une liste. Il s'avère que Python ne se soucie pas de `__str__` à cette fin.

Au lieu de cela, il recherche une méthode différente, `__repr__` , et si *ce n'est* pas trouvé, il retombe

sur la "chose hexadécimale". [2]

Donc, vous dites que je dois faire deux méthodes pour faire la même chose? Une pour quand je veux `print` ma carte seule et une autre quand elle est dans une sorte de conteneur?

Non, mais d'abord, regardons ce que *serait* notre classe si nous implémentions les méthodes `__str__` et `__repr__` :

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (S)" % (card_name, self.suit)

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (R)" % (card_name, self.suit)
```

Ici, l'implémentation des deux méthodes `__str__` et `__repr__` sont exactement les mêmes, sauf que pour différencier les deux méthodes, `(S)` est ajouté aux chaînes renvoyées par `__str__` et `(R)` est ajouté aux chaînes renvoyées par `__repr__` .

Notez que, tout comme notre méthode `__str__` , `__repr__` n'accepte aucun argument et **renvoie** une chaîne.

Nous pouvons voir maintenant quelle méthode est responsable de chaque cas:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)

my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]

print(my_hand)           # [Ace of Spades (R), 4 of Clubs (R), 6 of Hearts (R)]

print(ace_of_spades)     # Ace of Spades (S)
```

Comme nous l'avons vu, la méthode `__str__` été appelée lors de l' `print` notre instance `Card` et la méthode `__repr__` été appelée lorsque nous avons transmis *une liste de nos instances* à `print` .

À ce stade , il est intéressant de souligner que , tout comme nous pouvons créer explicitement une chaîne à partir d' une instance de classe personnalisée en utilisant `str()` comme nous l'avons fait précédemment, nous pouvons aussi créer explicitement une **représentation de chaîne** de notre classe avec une fonction intégrée appelée `repr()` .

Par exemple:

```
str_card = str(four_of_clubs)
```

```
print(str_card)                      # 4 of Clubs (S)

repr_card = repr(four_of_clubs)
print(repr_card)                     # 4 of Clubs (R)
```

De plus, si elle est définie, nous *pourrions* appeler les méthodes directement (même si cela semble peu clair et inutile):

```
print(four_of_clubs.__str__())     # 4 of Clubs (S)

print(four_of_clubs.__repr__())    # 4 of Clubs (R)
```

À propos de ces fonctions dupliquées ...

Les développeurs Python ont réalisé, dans le cas où vous vouliez que des chaînes identiques soient renvoyées par `str()` et `repr()` vous pourriez avoir à dupliquer de manière fonctionnelle des méthodes - quelque chose que personne n'aime.

Au lieu de cela, il existe un mécanisme pour éliminer ce besoin. Un que je vous ai fait passer jusqu'à ce point. Il s'avère que si une classe implémente la méthode `__repr__` *mais pas* la méthode `__str__` , et que vous passez une instance de cette classe à `str()` (implicitement ou explicitement), Python utilisera votre implémentation `__repr__` .

Donc, pour être clair, considérez la version suivante de la classe `Card` :

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)
```

Notez que cette version implémente *uniquement* la méthode `__repr__` . Néanmoins, les appels à `str()` donnent lieu à la version conviviale:

```
print(six_of_hearts)            # 6 of Hearts  (implicit conversion)
print(str(six_of_hearts))       # 6 of Hearts  (explicit conversion)
```

comme le font les appels à `repr()` :

```
print([six_of_hearts])          #[6 of Hearts] (implicit conversion)
print(repr(six_of_hearts))      # 6 of Hearts  (explicit conversion)
```

Résumé

Pour que vous puissiez autoriser vos instances de classe à "se montrer" de manière conviviale, vous devez envisager d'implémenter au moins la méthode `__repr__` votre classe. Si la mémoire est `__repr__` Raymond Hettinger a dit lors d'une conversation que la mise en œuvre des classes avec `__repr__` est l'une des premières choses qu'il cherche à faire lors de la révision du code Python. La quantité d'informations que vous *auriez* pu ajouter aux instructions de débogage, aux rapports d'erreur ou aux fichiers journaux avec une méthode simple est écrasante par rapport aux informations dérisoires et souvent peu utiles (type, id) fournies par défaut.

Si vous souhaitez *des* représentations *différentes* pour, par exemple, à l'intérieur d'un conteneur, vous devez implémenter les méthodes `__repr__` et `__str__` . (Plus d'informations sur la manière d'utiliser ces deux méthodes différemment ci-dessous).

Les deux méthodes sont implémentées, style eval-round-trip __repr __ ()

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    # Called when instance is converted to a string via str()
    # Examples:
    #   print(card1)
    #   print(str(card1)
    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)

    # Called when instance is converted to a string via repr()
    # Examples:
    #   print([card1, card2, card3])
    #   print(repr(card1))
    def __repr__(self):
        return "Card(%s, %d)" % (self.suit, self.pips)
```

Lire Représentations de chaîne des instances de classe: méthodes __str__ et __repr__ en ligne: https://riptutorial.com/fr/python/topic/4845/representations-de-chaine-des-instances-de-classe--methodes---str---et---repr--

Chapitre 172: Réseau Python

Remarques

Exemple de socket client Python (très) basique

Examples

Le plus simple exemple client-serveur de socket Python

Du côté serveur:

```
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8089))
serversocket.listen(5) # become a server socket, maximum 5 connections

while True:
    connection, address = serversocket.accept()
    buf = connection.recv(64)
    if len(buf) > 0:
        print(buf)
    break
```

Côté client:

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')
```

Commencez par exécuter SocketServer.py et assurez-vous que le serveur est prêt à écouter / recevoir sth. Le client envoie alors des informations au serveur. Après que le serveur a reçu sth, il se termine

Création d'un serveur HTTP simple

Pour partager des fichiers ou héberger des sites Web simples (http et javascript) sur votre réseau local, vous pouvez utiliser le module SimpleHTTPServer intégré de Python. Python devrait être dans votre variable Path. Accédez au dossier contenant vos fichiers et tapez:

Pour `python 2` :

```
$ python -m SimpleHTTPServer <portnumber>
```

Pour `python 3` :

```
$ python3 -m http.server <portnumber>
```

Si le numéro de port n'est pas indiqué, `8000` est le port par défaut. Donc, le résultat sera:

> Servant HTTP sur le port 0.0.0.0 8000 ...

Vous pouvez accéder à vos fichiers via n'importe quel périphérique connecté au réseau local en tapant `http://hostipaddress:8000/` .

`hostipaddress` est votre adresse IP locale qui commence probablement par `192.168.xx`

Pour terminer le module, appuyez simplement sur `ctrl+c`.

Créer un serveur TCP

Vous pouvez créer un serveur TCP à l'aide de la bibliothèque `socketserver` . Voici un serveur d'écho simple.

Du côté serveur

```
from sockerserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('connection from:', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    server = TCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Côté client

```
from socket import socket, AF_INET, SOCK_STREAM
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 5000))
sock.send(b'Monty Python')
sock.recv(8192)  # returns b'Monty Python'
```

`socketserver` facilite la création de serveurs TCP simples. Cependant, vous devez savoir que, par défaut, les serveurs sont à thread unique et ne peuvent servir qu'un client à la fois. Si vous souhaitez gérer plusieurs clients, instanciez à la place un `ThreadingTCPServer` .

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Création d'un serveur UDP

Un serveur UDP est facilement créé à l'aide de la bibliothèque `socketserver` .

un simple serveur de temps:

```
import time
from socketserver import BaseRequestHandler, UDPServer

class CtimeHandler(BaseRequestHandler):
    def handle(self):
    print('connection from: ', self.client_address)
    # Get message and client socket
    msg, sock = self.request
    resp = time.ctime()
    sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    server = UDPServer(('', 5000), CtimeHandler)
    server.serve_forever()
```

Essai:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> sock = socket(AF_INET, SOCK_DGRAM)
>>> sick.sendto(b'', ('localhost', 5000))
0
>>> sock.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 5000))
```

Démarrez Simple HttpServer dans un thread et ouvrez le navigateur

Utile si votre programme produit des pages Web en cours de route.

```
from http.server import HTTPServer, CGIHTTPRequestHandler
import webbrowser
import threading

def start_server(path, port=8000):
    '''Start a simple webserver serving path on port'''
    os.chdir(path)
    httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
    httpd.serve_forever()

# Start the server in a new thread
port = 8000
daemon = threading.Thread(name='daemon_server',
                          target=start_server,
                          args=('.', port)
daemon.setDaemon(True) # Set as a daemon so it will be killed once the main thread is dead.
daemon.start()

# Open the web browser
webbrowser.open('http://localhost:{}'.format(port))
```

Lire Réseau Python en ligne: https://riptutorial.com/fr/python/topic/1309/reseau-python

Chapitre 173: Sécurité et cryptographie

Introduction

Python, l'un des langages les plus populaires en matière de sécurité informatique et réseau, présente un grand potentiel en matière de sécurité et de cryptographie. Cette rubrique traite des fonctionnalités et des implémentations cryptographiques de Python, de ses utilisations dans la sécurité informatique et réseau aux algorithmes de hachage et de chiffrement / déchiffrement.

Syntaxe

- hashlib.new (nom)
- hashlib.pbkdf2_hmac (nom, mot de passe, sel, tours, dklen = Aucun)

Remarques

De nombreuses méthodes de `hashlib` nécessitent que vous passiez des valeurs interprétables en tant que tampons d'octets, plutôt que des chaînes. C'est le cas pour `hashlib.new().update()` ainsi que `hashlib.pbkdf2_hmac` . Si vous avez une chaîne, vous pouvez la convertir en un tampon d'octets en ajoutant le caractère `b` au début de la chaîne:

```
  "This is a string"
 b"This is a buffer of bytes"
```

Examples

Calcul d'un résumé de message

Le module `hashlib` permet de créer des générateurs de résumé de message via la `new` méthode. Ces générateurs transformeront une chaîne arbitraire en un condensé de longueur fixe:

```
import hashlib

h = hashlib.new('sha256')
h.update(b'Nobody expects the Spanish Inquisition.')
h.digest()
# ==>
b'.\xdf\xda\xdaVR[\x12\x90\xff\x16\xfb\x17D\xcf\xb4\x82\xdd)\x14\xff\xbc\xb6Iy\x0c\x0eX\x9eF-
='
```

Notez que vous pouvez appeler `update` un nombre arbitraire de fois avant d'appeler `digest` ce qui est utile pour hacher un bloc de fichier volumineux par bloc. Vous pouvez également obtenir le résumé au format hexadécimal en utilisant `hexdigest` :

```
h.hexdigest()
```

```
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d'
```

Algorithmes de hachage disponibles

`hashlib.new` nécessite le nom d'un algorithme lorsque vous l'appelez pour produire un générateur. Pour savoir quels algorithmes sont disponibles dans l'interpréteur Python actuel, utilisez `hashlib.algorithms_available` :

```
import hashlib
hashlib.algorithms_available
# ==> {'sha256', 'DSA-SHA', 'SHA512', 'SHA224', 'dsaWithSHA', 'SHA', 'RIPEMD160', 'ecdsa-with-
SHA1', 'sha1', 'SHA384', 'md5', 'SHA1', 'MD5', 'MD4', 'SHA256', 'sha384', 'md4', 'ripemd160',
'sha224', 'sha512', 'DSA', 'dsaEncryption', 'sha', 'whirlpool'}
```

La liste retournée variera selon la plate-forme et l'interprète; assurez-vous de vérifier que votre algorithme est disponible.

Il y a aussi quelques algorithmes qui sont *garantis* pour être disponibles sur toutes les plateformes et les interprètes, qui sont disponibles à l' aide `hashlib.algorithms_guaranteed` :

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

Hachage de mot de passe sécurisé

L' algorithme PBKDF2 exposé par le module `hashlib` peut être utilisé pour effectuer un hachage sécurisé des mots de passe. Bien que cet algorithme ne puisse pas empêcher les attaques par force brute afin de récupérer le mot de passe original à partir du hachage stocké, cela rend ces attaques très coûteuses.

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 peut fonctionner avec n'importe quel algorithme de résumé, l'exemple ci-dessus utilise SHA256 qui est généralement recommandé. Le sel aléatoire doit être stocké avec le mot de passe haché, vous en aurez de nouveau besoin pour comparer un mot de passe saisi au hachage stocké. Il est essentiel que chaque mot de passe soit haché avec un sel différent. En ce qui concerne le nombre de tours, il est recommandé de le définir aussi haut que possible pour votre application .

Si vous voulez le résultat en hexadécimal, vous pouvez utiliser le module `binascii` :

```
import binascii
hexhash = binascii.hexlify(hash)
```

Note : Bien que PBKDF2 ne soit pas mauvais, bcrypt et surtout scrypt sont considérés plus forts

contre les attaques en force. Aucune des deux ne fait partie de la bibliothèque standard Python pour le moment.

Hachage de fichiers

Un hachage est une fonction qui convertit une séquence d'octets de longueur variable en une séquence de longueur fixe. Le hachage de fichiers peut être avantageux pour de nombreuses raisons. Les hachages peuvent être utilisés pour vérifier si deux fichiers sont identiques ou vérifier que le contenu d'un fichier n'a pas été corrompu ou modifié.

Vous pouvez utiliser `hashlib` pour générer un hachage pour un fichier:

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

Pour les fichiers plus volumineux, un tampon de longueur fixe peut être utilisé:

```
import hashlib
SIZE = 65536
hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    buffer = f.read(SIZE)
    while len(buffer) > 0:
        hasher.update(buffer)
        buffer = f.read(SIZE)
print(hasher.hexdigest())
```

Chiffrement symétrique avec pycrypto

La fonctionnalité de cryptage intégrée de Python se limite actuellement au hachage. Le cryptage nécessite un module tiers tel que pycrypto . Par exemple, il fournit l' algorithme AES qui est considéré comme étant à la pointe de la technologie pour le chiffrement symétrique. Le code suivant chiffrera un message donné à l'aide d'une phrase secrète:

```
import hashlib
import math
import os

from Crypto.Cipher import AES

IV_SIZE = 16    # 128 bit, fixed for the AES algorithm
KEY_SIZE = 32   # 256 bit meaning AES-256, can also be 128 or 192 bits
SALT_SIZE = 16  # This size is arbitrary

cleartext = b'Lorem ipsum'
password = b'highly secure encryption password'
```

```
salt = os.urandom(SALT_SIZE)
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                              dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]

encrypted = salt + AES.new(key, AES.MODE_CFB, iv).encrypt(cleartext)
```

L'algorithme AES prend trois paramètres: la clé de chiffrement, le vecteur d'initialisation (IV) et le message réel à chiffrer. Si vous avez une clé AES générée aléatoirement, vous pouvez l'utiliser directement et simplement générer un vecteur d'initialisation aléatoire. Une phrase secrète n'a pas la bonne taille, cependant, il ne serait pas recommandé de l'utiliser directement, étant donné qu'elle n'est pas vraiment aléatoire et qu'elle a donc peu d'entropie. Au lieu de cela, nous utilisons l' implémentation intégrée de l'algorithme PBKDF2 pour générer un vecteur d'initialisation de 128 bits et une clé de chiffrement de 256 bits à partir du mot de passe.

Notez le sel aléatoire qui est important d'avoir un vecteur d'initialisation et une clé différents pour chaque message chiffré. Cela garantit notamment que deux messages égaux ne produiront pas un texte chiffré identique, mais empêchera également les attaquants de réutiliser le travail passé en devinant une phrase secrète sur les messages chiffrés avec une autre phrase secrète. Ce sel doit être stocké avec le message crypté afin de dériver le même vecteur d'initialisation et la même clé pour le décryptage.

Le code suivant déchiffrera à nouveau notre message:

```
salt = encrypted[0:SALT_SIZE]
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                              dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]
cleartext = AES.new(key, AES.MODE_CFB, iv).decrypt(encrypted[SALT_SIZE:])
```

Génération de signatures RSA à l'aide de pycrypto

RSA peut être utilisé pour créer une signature de message. Une signature valide ne peut être générée qu'avec un accès à la clé RSA privée, mais la validation est possible avec simplement la clé publique correspondante. Aussi longtemps que l'autre partie connaît votre clé publique, elle peut vérifier le message à signer par vous et inchangé - une approche utilisée par exemple pour le courrier électronique. Actuellement, un module tiers tel que pycrypto est requis pour cette fonctionnalité.

```
import errno

from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5

message = b'This message is from me, I promise.'

try:
    with open('privkey.pem', 'r') as f:
        key = RSA.importKey(f.read())
```

```
except IOError as e:
    if e.errno != errno.ENOENT:
        raise
    # No private key, generate a new one. This can take a few seconds.
    key = RSA.generate(4096)
    with open('privkey.pem', 'wb') as f:
        f.write(key.exportKey('PEM'))
    with open('pubkey.pem', 'wb') as f:
        f.write(key.publickey().exportKey('PEM'))

hasher = SHA256.new(message)
signer = PKCS1_v1_5.new(key)
signature = signer.sign(hasher)
```

La vérification de la signature fonctionne de la même manière mais utilise la clé publique plutôt que la clé privée:

```
with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
hasher = SHA256.new(message)
verifier = PKCS1_v1_5.new(key)
if verifier.verify(hasher, signature):
    print('Nice, the signature is valid!')
else:
    print('No, the message was signed with the wrong private key or modified')
```

Remarque : Les exemples ci-dessus utilisent l'algorithme de signature PKCS # 1 v1.5, qui est très courant. pycrypto implémente également le nouvel algorithme PSS PKCS # 1, en remplaçant `PKCS1_v1_5` par `PKCS1_PSS` dans les exemples qui devraient fonctionner si vous souhaitez utiliser celui-ci. Actuellement, il semble y avoir peu de raisons de l'utiliser .

Chiffrement asymétrique RSA avec pycrypto

Le chiffrement asymétrique présente l'avantage de pouvoir chiffrer un message sans échanger une clé secrète avec le destinataire du message. L'expéditeur doit simplement connaître la clé publique du destinataire, ce qui permet de chiffrer le message de telle sorte que seul le destinataire désigné (qui possède la clé privée correspondante) puisse le déchiffrer. Actuellement, un module tiers tel que pycrypto est requis pour cette fonctionnalité.

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message = b'This is a very secret message.'

with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
encrypted = cipher.encrypt(message)
```

Le destinataire peut déchiffrer le message s'il dispose de la clé privée appropriée:

```
with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
```

```
cipher = PKCS1_OAEP.new(key)
decrypted = cipher.decrypt(encrypted)
```

Remarque : les exemples ci-dessus utilisent le schéma de chiffrement PKCS # 1 OAEP. pycrypto implémente également le schéma de cryptage PKCS # 1 v1.5, celui-ci n'est cependant pas recommandé pour les nouveaux protocoles en raison de mises en garde connues .

Lire Sécurité et cryptographie en ligne: https://riptutorial.com/fr/python/topic/2598/securite-et-cryptographie

Chapitre 174: Sérialisation des données

Syntaxe

- unpickled_string = pickle.loads (chaîne)
- unpickled_string = pickle.load (objet_fichier)
- pickled_string = pickle.dumps ([('', 'cmplx'), {('objet',): Aucun}], pickle.HIGHEST_PROTOCOL)
- pickle.dump (('', 'cmplx'), {('objet',): aucun}], objet_fichier, pickle.HIGHEST_PROTOCOL)
- unjsoned_string = json.loads (chaîne)
- unjsoned_string = json.load (objet_fichier)
- jsoned_string = json.dumps (('a', 'b', 'c', [1, 2, 3]))
- json.dump (('a', 'b', 'c', [1, 2, 3]), objet_fichier)

Paramètres

Paramètre	Détails
`protocol`	En utilisant `pickle` ou `cPickle` , c'est la méthode utilisée pour sérialiser / désérialiser les objets. Vous voulez probablement utiliser `pickle.HIGHEST_PROTOCOL` ici, ce qui signifie la méthode la plus récente.

Remarques

Pourquoi utiliser JSON?

- Prise en charge multilingue
- Lisible par l'homme
- Contrairement au cornichon, il ne risque pas d'exécuter du code arbitraire

Pourquoi ne pas utiliser JSON?

- Ne prend pas en charge les types de données Pythonic
- Les clés dans les dictionnaires ne doivent pas être autres que les types de données de chaîne.

Pourquoi Pickle?

- Un excellent moyen de sérialiser Pythonic (tuples, fonctions, classes)
- Les clés dans les dictionnaires peuvent être de tout type de données.

Pourquoi pas Pickle?

- La prise en charge de plusieurs langues est manquante
- Il n'est pas sûr de charger des données arbitraires

Examples

Sérialisation en utilisant JSON

JSON est une méthode multilingue, largement utilisée pour sérialiser les données

Types de données pris en charge: *int* , *float* , *boolean* , *string* , *list* et *dict* . Voir -> Wiki JSON pour plus

Voici un exemple démontrant l'utilisation de **base** de **JSON** : -

```
import json

families = (['John'], ['Mark', 'David', {'name': 'Avraham'}])

# Dumping it into string
json_families = json.dumps(families)
# [["John"], ["Mark", "David", {"name": "Avraham"}]]

# Dumping it to file
with open('families.json', 'w') as json_file:
    json.dump(families, json_file)

# Loading it from string
json_families = json.loads(json_families)

# Loading it from file
with open('families.json', 'r') as json_file:
    json_families = json.load(json_file)
```

Voir le module JSON pour des informations détaillées sur JSON.

Sérialisation à l'aide de Pickle

Voici un exemple démontrant l'utilisation de **base** de **pickle** : -

```
# Importing pickle
try:
    import cPickle as pickle  # Python 2
except ImportError:
    import pickle  # Python 3

# Creating Pythonic object:
class Family(object):
    def __init__(self, names):
        self.sons = names

    def __str__(self):
        return ' '.join(self.sons)

my_family = Family(['John', 'David'])

# Dumping to string
pickle_data = pickle.dumps(my_family, pickle.HIGHEST_PROTOCOL)
```

```
# Dumping to file
with open('family.p', 'w') as pickle_file:
    pickle.dump(families, pickle_file, pickle.HIGHEST_PROTOCOL)

# Loading from string
my_family = pickle.loads(pickle_data)

# Loading from file
with open('family.p', 'r') as pickle_file:
    my_family = pickle.load(pickle_file)
```

Voir Pickle pour des informations détaillées sur Pickle.

AVERTISSEMENT : La documentation officielle de pickle indique clairement qu'il n'y a aucune garantie de sécurité. Ne chargez aucune donnée dont vous ne connaissez pas l'origine.

Lire Sérialisation des données en ligne: https://riptutorial.com/fr/python/topic/3347/serialisation-des-donnees

Chapitre 175: Sérialisation des données de pickle

Syntaxe

- pickle.dump (objet, fichier, protocole) #Pour sérialiser un objet
- pickle.load (file) #Pour désérialiser un objet
- pickle.dumps (objet, protocole) # Pour sérialiser un objet en octets
- pickle.loads (buffer) # Pour désérialiser un objet à partir d'octets

Paramètres

Paramètre	Détails
objet	L'objet à stocker
fichier	Le fichier ouvert qui contiendra l'objet
protocole	Le protocole utilisé pour décrocher l'objet (paramètre facultatif)
tampon	Un objet octets contenant un objet sérialisé

Remarques

Types de picklables

Les objets suivants sont picklable.

- `None` , `True` et `False`
- nombres (de tous types)
- cordes (de tous types)
- `tuple` s, `list` s, `set` s et `dict` contenant uniquement des objets picklable
- fonctions définies au niveau supérieur d'un module
- fonctions intégrées
- classes définies au niveau supérieur d'un module
    - des instances de telles classes dont le `__dict__` ou le résultat de l'appel de `__getstate__()` est picklable (voir les documents officiels pour plus de détails).

Basé sur la documentation officielle de Python .

`pickle` et sécurité

Le module de pickle n'est **pas sécurisé** . Il ne doit pas être utilisé lors de la réception de données sérialisées provenant d'une partie non fiable, par exemple sur Internet.

Examples

Utiliser Pickle pour sérialiser et désérialiser un objet

Le module `pickle` implémente un algorithme pour transformer un objet Python arbitraire en une série d'octets. Ce processus est également appelé **sérialisation de** l'objet. Le flux d'octets représentant l'objet peut alors être transmis ou stocké, puis reconstruit pour créer un nouvel objet avec les mêmes caractéristiques.

Pour le code le plus simple, nous utilisons les fonctions `dump()` et `load()` .

Pour sérialiser l'objet

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

Désérialiser l'objet

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

Utilisation d'objets pickle et byte

Il est également possible de sérialisation dans et désérialiser à partir d'objets d'octets, en utilisant

les `dumps` et `loads` fonction, qui sont équivalentes à `dump` et `load` .

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) is bytes

deserialized_data = pickle.loads(serialized_data)
# deserialized_data == data
```

Personnaliser les données marinées

Certaines données ne peuvent pas être décapées. Les autres données ne doivent pas être décapées pour d'autres raisons.

Ce qui sera décapé peut être défini dans la méthode `__getstate__` . Cette méthode doit renvoyer quelque chose qui est picklable.

Sur le côté opposé, `__setstate__` : il recevra ce que `__getstate__` créé et doit initialiser l'objet.

```
class A(object):
    def __init__(self, important_data):
        self.important_data = important_data

        # Add data which cannot be pickled:
        self.func = lambda: 7

        # Add data which should never be pickled, because it expires quickly:
        self.is_up_to_date = False

    def __getstate__(self):
        return [self.important_data] # only this is needed

    def __setstate__(self, state):
        self.important_data = state[0]

        self.func = lambda: 7  # just some hard-coded unpicklable function

        self.is_up_to_date = False  # even if it was before pickling
```

Maintenant, cela peut être fait:

```
>>> a1 = A('very important')
>>>
>>> s = pickle.dumps(a1)  # calls a1.__getstate__()
>>>
>>> a2 = pickle.loads(s)  # calls a1.__setstate__(['very important'])
>>> a2
<__main__.A object at 0x0000000002742470>
>>> a2.important_data
'very important'
>>> a2.func()
7
```

L'implémentation ici affiche une liste avec une valeur: `[self.important_data]` . C'était juste un exemple, `__getstate__` aurait pu retourner tout ce qui est picklable, tant que `__setstate__` sait comment oppoisite. Une bonne alternative est un dictionnaire de toutes les valeurs:

`{'important_data': self.important_data}` .

Le constructeur n'est pas appelé! Notez que dans l'exemple précédent, l'instance `a2` été créée dans `pickle.loads` sans jamais appeler `A.__init__` , donc `A.__setstate__` devait initialiser tout ce que `__init__` aurait initialisé s'il était appelé.

Lire Sérialisation des données de pickle en ligne:
https://riptutorial.com/fr/python/topic/2606/serialisation-des-donnees-de-pickle

Chapitre 176: Serveur HTTP Python

Examples

Exécution d'un serveur HTTP simple

Python 2.x 2.3

```
python -m SimpleHTTPServer 9000
```

Python 3.x 3.0

```
python -m http.server 9000
```

L'exécution de cette commande sert les fichiers du répertoire en cours sur le port `9000` .

Si aucun argument n'est fourni comme numéro de port, le serveur s'exécutera sur le port `8000` par défaut.

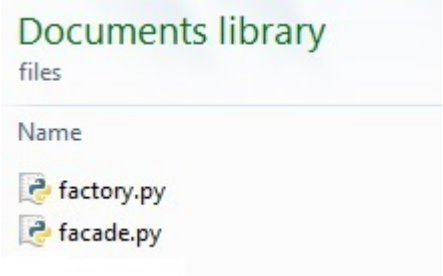
L' `-m` recherche `sys.path` pour le fichier `.py` correspondant à exécuter en tant que module.

Si vous souhaitez ne servir que sur localhost, vous devrez écrire un programme Python personnalisé tel que:

```
import sys
import BaseHTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

HandlerClass = SimpleHTTPRequestHandler
ServerClass  = BaseHTTPServer.HTTPServer
Protocol     = "HTTP/1.0"

if sys.argv[1:]:
   port = int(sys.argv[1])
else:
   port = 8000
server_address = ('127.0.0.1', port)

HandlerClass.protocol_version = Protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

Fichiers de service

En supposant que vous ayez le répertoire de fichiers suivant:



Vous pouvez configurer un serveur Web pour servir ces fichiers comme suit:

Python 2.x 2.3

```
import SimpleHTTPServer
import SocketServer

PORT = 8000

handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer(("localhost", PORT), handler)
print "Serving files at port {}".format(PORT)
httpd.serve_forever()
```

Python 3.x 3.0

```
import http.server
import socketserver

PORT = 8000

handler = http.server.SimpleHTTPRequestHandler
httpd = socketserver.TCPServer(("", PORT), handler)
print("serving at port", PORT)
httpd.serve_forever()
```

Le module `SocketServer` fournit les classes et les fonctionnalités permettant de configurer un serveur réseau.

`SocketServer` de `TCPServer` classe met en place un serveur en utilisant le protocole TCP. Le constructeur accepte un tuple représentant l'adresse du serveur (c'est-à-dire l'adresse IP et le port) et la classe qui gère les requêtes du serveur.

La `SimpleHTTPRequestHandler` classe du `SimpleHTTPServer` module permet les fichiers dans le répertoire en cours à desservir.

Enregistrez le script dans le même répertoire et exécutez-le.

Exécutez le serveur HTTP:

Python 2.x 2.3

python -m SimpleHTTPServer 8000

Python 3.x 3.0

python -m http.server 8000

L'indicateur '-m' recherchera 'sys.path' pour le fichier '.py' correspondant à exécuter en tant que module.

Ouvrez localhost: 8000 dans le navigateur, cela vous donnera les informations suivantes:

API programmatique de SimpleHTTPServer

Que se passe-t-il lorsque nous `python -m SimpleHTTPServer 9000` ?

Pour répondre à cette question, nous devons comprendre la construction de SimpleHTTPServer (https://hg.python.org/cpython/file/2.7/Lib/SimpleHTTPServer.py) et BaseHTTPServer (https://hg.python.org/cpython/file /2.7/Lib/BaseHTTPServer.py) .

Tout d'abord, Python appelle le module `SimpleHTTPServer` avec `9000` comme argument. En observant maintenant le code SimpleHTTPServer,

```
def test(HandlerClass = SimpleHTTPRequestHandler,
         ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)


if __name__ == '__main__':
    test()
```

La fonction de test est appelée après les gestionnaires de requêtes et ServerClass. Maintenant, BaseHTTPServer.test est invoqué

```
def test(HandlerClass = BaseHTTPRequestHandler,
         ServerClass = HTTPServer, protocol="HTTP/1.0"):
"""Test the HTTP request handler class.

This runs an HTTP server on port 8000 (or the first command line
argument).

"""

if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000
```

```
server_address = ('', port)

HandlerClass.protocol_version = protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

D'où le numéro de port, que l'utilisateur a passé en argument est analysé et lié à l'adresse de l'hôte. D'autres étapes de base de la programmation des sockets avec un port et un protocole donnés sont effectuées. Enfin, le serveur de socket est lancé.

Voici un aperçu de base de l'héritage de la classe SocketServer vers d'autres classes:

```
    +------------+
    | BaseServer |
    +------------+
          |
          v
    +-----------+        +------------------+
    | TCPServer |------->| UnixStreamServer |
    +-----------+        +------------------+
          |
          v
    +-----------+        +--------------------+
    | UDPServer |------->| UnixDatagramServer |
    +-----------+        +--------------------+
```

Les liens https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py et https://hg.python.org/cpython/file/2.7/Lib/SocketServer.py sont utiles pour trouver d'autres informations. information.

Gestion de base de GET, POST, PUT en utilisant BaseHTTPRequestHandler

```
# from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer # python2
from http.server import BaseHTTPRequestHandler, HTTPServer # python3
class HandleRequests(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        self._set_headers()
        self.wfile.write("received get request")

    def do_POST(self):
        '''Reads post request body'''
        self._set_headers()
        content_len = int(self.headers.getheader('content-length', 0))
        post_body = self.rfile.read(content_len)
        self.wfile.write("received post request:<br>{}".format(post_body))

    def do_PUT(self):
        self.do_POST()
```

```
host = ''
port = 80
HTTPServer((host, port), HandleRequests).serve_forever()
```

Exemple de sortie utilisant `curl` :

```
$ curl http://localhost/
received get request%


$ curl -X POST http://localhost/
received post request:<br>%


$ curl -X PUT http://localhost/
received post request:<br>%


$ echo 'hello world' | curl --data-binary @- http://localhost/
received post request:<br>hello world
```

Lire Serveur HTTP Python en ligne: https://riptutorial.com/fr/python/topic/4247/serveur-http-python

Chapitre 177: setup.py

Paramètres

Paramètre	Usage
`name`	Nom de votre distribution.
`version`	Chaîne de version de votre distribution.
`packages`	Liste des packages Python (c'est-à-dire des répertoires contenant des modules) à inclure. Cela peut être spécifié manuellement, mais un appel à `setuptools.find_packages()` est généralement utilisé à la place.
`py_modules`	Liste des modules Python de niveau supérieur (c'est-à-dire des fichiers `.py` uniques) à inclure.

Remarques

Pour plus d'informations sur les emballages en python, voir:

introduction

Pour rédiger des paquets officiels, il existe un guide d'utilisation des emballages .

Examples

But de setup.py

Le script de configuration est le centre de toutes les activités de création, de distribution et d'installation des modules utilisant les Distutils. Son but est l'installation correcte du logiciel.

Si tout ce que vous voulez faire est de distribuer un module appelé foo, contenu dans un fichier foo.py, alors votre script de configuration peut être aussi simple que cela:

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Pour créer une distribution source pour ce module, vous devez créer un script d'installation, setup.py, contenant le code ci-dessus, et exécuter cette commande à partir d'un terminal:

```
python setup.py sdist
```

sdist va créer un fichier d'archive (par exemple, une archive tar sur Unix, un fichier ZIP sous Windows) contenant votre script d'installation setup.py et votre module foo.py. Le fichier d'archive s'appellera foo-1.0.tar.gz (ou .zip) et décompressera dans un répertoire foo-1.0.

Si un utilisateur souhaite installer votre module foo, il lui suffit de télécharger foo-1.0.tar.gz (ou .zip), de le décompresser et, à partir du répertoire foo-1.0, de l'exécuter.

```
python setup.py install
```

Ajout de scripts de ligne de commande à votre package python

Les scripts de ligne de commande dans les packages python sont communs. Vous pouvez organiser votre paquet de telle manière que lorsqu'un utilisateur installe le paquet, le script sera disponible sur son chemin.

Si vous aviez le paquet de `greetings` qui avait le script de ligne de commande `hello_world.py` .

```
greetings/
   greetings/
      __init__.py
      hello_world.py
```

Vous pouvez exécuter ce script en exécutant:

```
python greetings/greetings/hello_world.py
```

Cependant, si vous souhaitez le lancer comme ceci:

```
hello_world.py
```

Vous pouvez y parvenir en ajoutant des `scripts` à votre `setup()` dans `setup.py` comme ceci:

```
from setuptools import setup
setup(
  name='greetings',
  scripts=['hello_world.py']
)
```

Lorsque vous installez le package de salutations maintenant, `hello_world.py` sera ajouté à votre chemin.

Une autre possibilité serait d'ajouter un point d'entrée:

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

De cette façon, il vous suffit de le lancer comme:

```
greetings
```

Utilisation des métadonnées du contrôle de code source dans setup.py

`setuptools_scm` est un paquet officiellement béni qui peut utiliser les métadonnées Git ou Mercurial pour déterminer le numéro de version de votre paquet et trouver les paquets Python et les données de paquet à inclure.

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

Cet exemple utilise les deux fonctionnalités; pour utiliser uniquement les métadonnées SCM pour la version, remplacez l'appel à `find_packages()` par votre liste de packages manuelle ou utilisez uniquement l'outil de recherche de package, supprimez `use_scm_version=True` .

Ajout d'options d'installation

Comme on l'a vu dans les exemples précédents, l'utilisation de base de ce script est la suivante:

```
python setup.py install
```

Mais il y a encore plus d'options, comme installer le paquet et avoir la possibilité de changer le code et de le tester sans avoir à le réinstaller. Ceci est fait en utilisant:

```
python setup.py develop
```

Si vous souhaitez effectuer des actions spécifiques telles que la compilation d'une documentation *Sphinx* ou la construction d'un code *fortran* , vous pouvez créer votre propre option comme celle-ci:

```
cmdclasses = dict()

class BuildSphinx(Command):

    """Build Sphinx documentation."""

    description = 'Build Sphinx documentation'
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        import sphinx
        sphinx.build_main(['setup.py', '-b', 'html', './doc', './doc/_build/html'])
```

```
        sphinx.build_main(['setup.py', '-b', 'man', './doc', './doc/_build/man'])

cmdclasses['build_sphinx'] = BuildSphinx

setup(
...
cmdclass=cmdclasses,
)
```

`initialize_options` et `finalize_options` seront exécutés avant et après la fonction d' `run` , comme leur nom l'indique.

Après cela, vous pourrez appeler votre option:

```
python setup.py build_sphinx
```

Lire setup.py en ligne: https://riptutorial.com/fr/python/topic/1444/setup-py

Chapitre 178: Similitudes dans la syntaxe, différences de sens: Python vs JavaScript

Introduction

Il arrive parfois que deux langages aient des significations différentes sur la même expression ou une syntaxe similaire. Lorsque les deux langues présentent un intérêt pour un programmeur, la clarification de ces points de bifurcation permet de mieux comprendre les deux langues dans leurs bases et leurs subtilités.

Examples

`in` avec des listes

```
2 in [2, 3]
```

En Python, ceci est évalué à True, mais en JavaScript à false. En effet, dans Python en vérifie si une valeur est contenue dans une liste, alors 2 est dans [2, 3] comme premier élément. Dans JavaScript, est utilisé avec des objets et vérifie si un objet contient la propriété avec le nom exprimé par la valeur. Donc, JavaScript considère [2, 3] comme un objet ou une carte clé-valeur comme ceci:

```
{'0': 2, '1': 3}
```

et vérifie s'il a une propriété ou une clé '2'. Entier 2 est converti en silence en chaîne '2'.

Lire Similitudes dans la syntaxe, différences de sens: Python vs JavaScript en ligne: https://riptutorial.com/fr/python/topic/10766/similitudes-dans-la-syntaxe--differences-de-sens--python-vs-javascript

Chapitre 179: Sockets et cryptage / décryptage de messages entre le client et le serveur

Introduction

La cryptographie est utilisée à des fins de sécurité. Il n'y a pas beaucoup d'exemples de cryptage / décryptage en Python en utilisant le cryptage MODE CTR IDEA. **But de cette documentation:**

Étendre et implémenter le schéma de signature numérique RSA dans les communications entre stations. Utiliser Hashing pour l'intégrité du message, c'est SHA-1. Produire un protocole de transport de clé simple. Crypter la clé avec le cryptage IDEA. Le mode de chiffrement de bloc est le mode compteur

Remarques

Langue utilisée: Python 2.7 (lien de téléchargement: https://www.python.org/downloads/)

Bibliothèque utilisée:

* **PyCrypto** (lien de téléchargement: https://pypi.python.org/pypi/pycrypto)

* **PyCryptoPlus** (lien de téléchargement: https://github.com/doegox/python-cryptoplus)

Installation de la bibliothèque:

PyCrypto: Décompressez le fichier. Allez dans le répertoire et ouvrez le terminal pour Linux (alt + ctrl + t) et CMD (shift + clic droit + sélection de l'invite de commande ouverte ici) pour Windows. Après cela, écrivez python setup.py install (Assurez-vous que Python Environment est correctement configuré dans Windows)

PyCryptoPlus: Identique à la dernière bibliothèque.

Tâches Implémentation: La tâche est séparée en deux parties. L'un est un processus de prise de contact et l'autre un processus de communication. Configuration du socket:

- Comme la création de clés publiques et privées ainsi que le hachage de la clé publique, nous devons configurer le socket maintenant. Pour configurer le socket, nous devons importer un autre module avec «socket d'importation» et connecter (pour le client) ou lier (pour le serveur) l'adresse IP et le port avec le socket venant de l'utilisateur.

  ----------Côté client----------

  ```
  server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
  host = raw_input("Server Address To Be Connected -> ")
  ```

```
port = int(input("Port of The Server -> "))
server.connect((host, port))
```

----------Du côté serveur---------

```
try:
#setting up socket
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server.bind((host,port))
server.listen(5)
except BaseException: print "-----Check Server Address or Port-----"
```

"Socket.AF_INET, socket.SOCK_STREAM" nous permettra d'utiliser la fonction accept () et les fondamentaux de la messagerie. Au lieu de cela, nous pouvons utiliser «socket.AF_INET, socket.SOCK_DGRAM» également, mais cette fois, nous devrons utiliser setblocking (value).

Processus de poignée de main:

- (CLIENT) La première tâche consiste à créer une clé publique et privée. Pour créer la clé privée et publique, nous devons importer certains modules. Ce sont: depuis l'importation Crypto Random et depuis l'importation Crypto.PublicKey RSA. Pour créer les clés, il faut écrire quelques lignes de codes simples:

```
random_generator = Random.new().read
        key = RSA.generate(1024,random_generator)
        public = key.publickey().exportKey()
```

random_generator est dérivé du module " **from Crypto import Random** ". La clé est dérivée de «à **partir de Crypto.PublicKey import RSA** » qui créera une clé privée de 1024 en générant des caractères aléatoires. Public exporte la clé publique de la clé privée précédemment générée.

- (CLIENT) Après avoir créé la clé publique et privée, nous devons hacher la clé publique pour envoyer au serveur en utilisant le hachage SHA-1. Pour utiliser le hachage SHA-1, nous devons importer un autre module en écrivant «import hashlib». Pour hacher la clé publique, nous avons écrit deux lignes de code:

```
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()
```

Hash_object et hex_digest sont nos variables. Après cela, le client enverra hex_digest et public au serveur et le serveur les vérifiera en comparant le hash obtenu du client et le nouveau hachage de la clé publique. Si le nouveau hachage et le hachage du client correspondent, il passera à la procédure suivante. Comme le public envoyé par le client est sous forme de chaîne, il ne pourra pas être utilisé comme clé du côté serveur. Pour empêcher cela et convertir la clé publique de chaîne en clé publique rsa, nous devons écrire `server_public_key = RSA.importKey(getpbk)` , ici getpbk est la clé publique du client.

- (SERVER) L'étape suivante consiste à créer une clé de session. Ici, j'ai utilisé le module "os"

pour créer une clé aléatoire "key = os.urandom (16)" qui nous donnera une clé de 16 bits et après cela, j'ai chiffré cette clé dans "AES.MODE_CTR" avec SHA-1:

```
 #encrypt CTR MODE session key
 en = AES.new(key_128,AES.MODE_CTR,counter = lambda:key_128) encrypto =
en.encrypt(key_128)
 #hashing sha1
 en_object = hashlib.sha1(encrypto)
 en_digest = en_object.hexdigest()
```

Donc, le en_digest sera notre clé de session.

- (SERVER) La dernière partie du processus de prise de contact consiste à chiffrer la clé publique obtenue à partir du client et la clé de session créée côté serveur.

```
 #encrypting session key and public key
 E = server_public_key.encrypt(encrypto,16)
```

Après le cryptage, le serveur enverra la clé au client sous forme de chaîne.

- (CLIENT) Après avoir obtenu la chaîne chiffrée de (clé publique et de session) du serveur, le client les déchiffrera à l'aide de la clé privée créée précédemment avec la clé publique. Comme le crypté (clé publique et clé de session) était sous forme de chaîne, nous devons maintenant le récupérer en tant que clé en utilisant eval (). Si le déchiffrement est effectué, le processus de prise de contact est également terminé, les deux parties confirmant qu'elles utilisent les mêmes clés. Déchiffrer:

```
 en = eval(msg)
 decrypt = key.decrypt(en)
 # hashing sha1
 en_object = hashlib.sha1(decrypt) en_digest = en_object.hexdigest()
```

J'ai utilisé le SHA-1 ici pour qu'il soit lisible dans la sortie.

Processus de communication:

Pour le processus de communication, nous devons utiliser la clé de session des deux côtés comme clé pour le cryptage MODE_CTR. Les deux parties vont chiffrer et déchiffrer les messages avec IDEA.MODE_CTR en utilisant la clé de session.

- (Cryptage) Pour le cryptage IDEA, nous avons besoin d'une clé de 16 bits et d'un compteur tel que pouvant être appelé. Le compteur est obligatoire dans MODE_CTR. La clé de session que nous avons chiffrée et hachée est désormais de 40, ce qui dépassera la clé de limite du chiffrement IDEA. Par conséquent, nous devons réduire la taille de la clé de session. Pour réduire, nous pouvons utiliser la chaîne de fonctions normale de python intégrée [valeur: valeur]. Où la valeur peut être n'importe quelle valeur en fonction du choix de l'utilisateur. Dans notre cas, j'ai fait “key [: 16]” où il faudra de 0 à 16 valeurs de la clé. Cette conversion pourrait être faite de plusieurs manières comme la clé [1:17] ou la clé [16:]. La prochaine partie consiste à créer une nouvelle fonction de chiffrement IDEA en écrivant IDEA.new () qui prendra trois arguments pour le traitement. Le premier argument sera KEY,

le deuxième argument sera le mode de cryptage IDEA (dans notre cas, IDEA.MODE_CTR) et le troisième argument sera le compteur = qui est une fonction appelable. Le compteur = contiendra une taille de chaîne qui sera retournée par la fonction. Pour définir le compteur =, il faut utiliser des valeurs raisonnables. Dans ce cas, j'ai utilisé la taille de la clé en définissant lambda. Au lieu d'utiliser lambda, nous pourrions utiliser Counter.Util qui génère une valeur aléatoire pour le compteur =. Pour utiliser Counter.Util, nous devons importer un module de compteur à partir de crypto. Par conséquent, le code sera:

```
ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
```

Une fois que vous avez défini «ideaEncrypt» comme variable de chiffrement IDEA, vous pouvez utiliser la fonction de chiffrement intégrée pour chiffrer n'importe quel message.

```
eMsg = ideaEncrypt.encrypt(whole)
#converting the encrypted message to HEXADECIMAL to readable eMsg =
eMsg.encode("hex").upper()
```

Dans ce segment de code, tout est le message à chiffrer et eMsg est le message chiffré. Après avoir chiffré le message, je l'ai converti en HEXADECIMAL pour le rendre lisible et upper () est la fonction intégrée pour rendre les caractères en majuscule. Après cela, ce message crypté sera envoyé à la station opposée pour décryptage.

- **(Décryptage)**

Pour déchiffrer les messages chiffrés, nous devrons créer une autre variable de chiffrement en utilisant les mêmes arguments et la même clé, mais cette fois-ci, la variable déchiffrera les messages chiffrés. Le code pour le même que la dernière fois. Cependant, avant de déchiffrer les messages, nous devons décoder le message en hexadécimal car, dans notre partie chiffrement, nous avons encodé le message chiffré en hexadécimal pour le rendre lisible. Par conséquent, le code entier sera:

```
decoded = newmess.decode("hex")
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
dMsg = ideaDecrypt.decrypt(decoded)
```

Ces processus seront effectués côté serveur et côté client pour le cryptage et le décryptage.

Examples

Implémentation côté serveur

```
import socket
import hashlib
import os
import time
import itertools
import threading
import sys
import Crypto.Cipher.AES as AES
```

```
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#server address and port number input from admin
host= raw_input("Server Address - > ")
port = int(input("Port - > "))
#boolean for checking server and port
check = False
done = False

def animate():
    for c in itertools.cycle(['....','.......','..........','............']):
        if done:
            break
        sys.stdout.write('\rCHECKING IP ADDRESS AND NOT USED PORT '+c)
        sys.stdout.flush()
        time.sleep(0.1)
    sys.stdout.write('\r -----SERVER STARTED. WAITING FOR CLIENT-----\n')
try:
    #setting up socket
    server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    server.bind((host,port))
    server.listen(5)
    check = True
except BaseException:
    print "-----Check Server Address or Port-----"
    check = False

if check is True:
    # server Quit
    shutdown = False
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True
#binding client and address
client,address = server.accept()
print ("CLIENT IS CONNECTED. CLIENT'S ADDRESS ->",address)
print ("\n-----WAITING FOR PUBLIC KEY & PUBLIC KEY HASH-----\n")

#client's message(Public Key)
getpbk = client.recv(2048)

#conversion of string to KEY
server_public_key = RSA.importKey(getpbk)

#hashing the public key in server side for validating the hash from client
hash_object = hashlib.sha1(getpbk)
hex_digest = hash_object.hexdigest()

if getpbk != "":
    print (getpbk)
    client.send("YES")
    gethash = client.recv(1024)
    print ("\n-----HASH OF PUBLIC KEY----- \n"+gethash)
if hex_digest == gethash:
    # creating session key
    key_128 = os.urandom(16)
    #encrypt CTR MODE session key
```

```
    en = AES.new(key_128,AES.MODE_CTR,counter = lambda:key_128)
    encrypto = en.encrypt(key_128)
    #hashing sha1
    en_object = hashlib.sha1(encrypto)
    en_digest = en_object.hexdigest()

    print ("\n-----SESSION KEY-----\n"+en_digest)

    #encrypting session key and public key
    E = server_public_key.encrypt(encrypto,16)
    print ("\n-----ENCRYPTED PUBLIC KEY AND SESSION KEY-----\n"+str(E))
    print ("\n-----HANDSHAKE COMPLETE-----")
    client.send(str(E))
    while True:
        #message from client
        newmess = client.recv(1024)
        #decoding the message from HEXADECIMAL to decrypt the ecrypted version of the message
only
        decoded = newmess.decode("hex")
        #making en_digest(session_key) as the key
        key = en_digest[:16]
        print ("\nENCRYPTED MESSAGE FROM CLIENT -> "+newmess)
        #decrypting message from the client
        ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
        dMsg = ideaDecrypt.decrypt(decoded)
        print ("\n**New Message**  "+time.ctime(time.time()) +" > "+dMsg+"\n")
        mess = raw_input("\nMessage To Client -> ")
        if mess != "":
            ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
            eMsg = ideaEncrypt.encrypt(mess)
            eMsg = eMsg.encode("hex").upper()
            if eMsg != "":
                print ("ENCRYPTED MESSAGE TO CLIENT-> " + eMsg)
            client.send(eMsg)
    client.close()
else:
    print ("\n-----PUBLIC KEY HASH DOESNOT MATCH-----\n")
```

Implémentation côté client

```
import time
import socket
import threading
import hashlib
import itertools
import sys
from Crypto import Random
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#animating loading
done = False
def animate():
    for c in itertools.cycle(['....','.......','..........','............']):
        if done:
            break
        sys.stdout.write('\rCONFIRMING CONNECTION TO SERVER '+c)
        sys.stdout.flush()
        time.sleep(0.1)
```

```
#public key and private key
random_generator = Random.new().read
key = RSA.generate(1024,random_generator)
public = key.publickey().exportKey()
private = key.exportKey()

#hashing the public key
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()

#Setting up socket
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

#host and port input user
host = raw_input("Server Address To Be Connected -> ")
port = int(input("Port of The Server -> "))
#binding the address and port
server.connect((host, port))
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True

def send(t,name,key):
    mess = raw_input(name + " : ")
    key = key[:16]
    #merging the message and the name
    whole = name+" : "+mess
    ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
    eMsg = ideaEncrypt.encrypt(whole)
    #converting the encrypted message to HEXADECIMAL to readable
    eMsg = eMsg.encode("hex").upper()
    if eMsg != "":
        print ("ENCRYPTED MESSAGE TO SERVER-> "+eMsg)
    server.send(eMsg)
def recv(t,key):
    newmess = server.recv(1024)
    print ("\nENCRYPTED MESSAGE FROM SERVER-> " + newmess)
    key = key[:16]
    decoded = newmess.decode("hex")
    ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
    dMsg = ideaDecrypt.decrypt(decoded)
    print ("\n**New Message From Server**  " + time.ctime(time.time()) + " : " + dMsg + "\n")

while True:
    server.send(public)
    confirm = server.recv(1024)
    if confirm == "YES":
        server.send(hex_digest)

    #connected msg
    msg = server.recv(1024)
    en = eval(msg)
    decrypt = key.decrypt(en)
    # hashing sha1
    en_object = hashlib.sha1(decrypt)
    en_digest = en_object.hexdigest()
```

```
    print ("\n-----ENCRYPTED PUBLIC KEY AND SESSION KEY FROM SERVER-----")
    print (msg)
    print ("\n-----DECRYPTED SESSION KEY-----")
    print (en_digest)
    print ("\n-----HANDSHAKE COMPLETE-----\n")
    alais = raw_input("\nYour Name -> ")

    while True:
        thread_send = threading.Thread(target=send,args=("------Sending Message------
",alais,en_digest))
        thread_recv = threading.Thread(target=recv,args=("------Recieving Message------
",en_digest))
        thread_send.start()
        thread_recv.start()

        thread_send.join()
        thread_recv.join()
        time.sleep(0.5)
    time.sleep(60)
    server.close()
```

Lire Sockets et cryptage / décryptage de messages entre le client et le serveur en ligne: https://riptutorial.com/fr/python/topic/8710/sockets-et-cryptage---decryptage-de-messages-entre-le-client-et-le-serveur

Chapitre 180: Sous-commandes CLI avec sortie d'aide précise

Introduction

Différentes manières de créer des sous-commandes comme dans `hg` ou `svn` avec l'interface de ligne de commande exacte et les résultats d'aide, comme indiqué dans la section Remarques.

L'analyse des arguments de la ligne de commande couvre un sujet plus large d'analyse des arguments.

Remarques

Différentes manières de créer des sous-commandes comme dans `hg` ou `svn` avec l'interface de ligne de commande affichée dans le message d'aide:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Examples

Façon native (pas de bibliothèques)

```
"""
usage: sub <command>

commands:

  status -  show status
  list   -  print list
"""

import sys

def check():
    print("status")
    return 0

if sys.argv[1:] == ['status']:
    sys.exit(check())
elif sys.argv[1:] == ['list']:
    print("list")
else:
    print(__doc__.strip())
```

Sortie sans arguments:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Avantages:

- pas de deps
- tout le monde devrait pouvoir lire cela
- contrôle complet du formatage de l'aide

argparse (formateur d'aide par défaut)

```
import argparse
import sys

def check():
    print("status")
    return 0

parser = argparse.ArgumentParser(prog="sub", add_help=False)
subparser = parser.add_subparsers(dest="cmd")

subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())
```

Sortie sans arguments:

```
usage: sub {status,list} ...

positional arguments:
  {status,list}
    status       show status
    list         print list
```

Avantages:

- est livré avec Python
- l'analyse des options est incluse

argparse (formateur d'aide personnalisée)

Version étendue de http://www.riptutorial.com/python/example/25282/argparse--default-help-formatter- cette sortie d'aide fixe.

```
import argparse
import sys

class CustomHelpFormatter(argparse.HelpFormatter):
    def _format_action(self, action):
        if type(action) == argparse._SubParsersAction:
            # inject new class variable for subcommand formatting
            subactions = action._get_subactions()
            invocations = [self._format_action_invocation(a) for a in subactions]
            self._subcommand_max_length = max(len(i) for i in invocations)

        if type(action) == argparse._SubParsersAction._ChoicesPseudoAction:
            # format subcommand help line
            subcommand = self._format_action_invocation(action) # type: str
            width = self._subcommand_max_length
            help_text = ""
            if action.help:
                help_text = self._expand_help(action)
            return "  {:{width}} -  {}\n".format(subcommand, help_text, width=width)

        elif type(action) == argparse._SubParsersAction:
            # process subcommand help section
            msg = '\n'
            for subaction in action._get_subactions():
                msg += self._format_action(subaction)
            return msg
        else:
            return super(CustomHelpFormatter, self)._format_action(action)


def check():
    print("status")
    return 0

parser = argparse.ArgumentParser(usage="sub <command>", add_help=False,
             formatter_class=CustomHelpFormatter)

subparser = parser.add_subparsers(dest="cmd")
subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# custom help messge
parser._positionals.title = "commands"

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
```

```
    sys.exit(check())
```

Sortie sans arguments:

```
usage: sub <command>

commands:

  status -  show status
  list   -  print list
```

Lire Sous-commandes CLI avec sortie d'aide précise en ligne:
https://riptutorial.com/fr/python/topic/7701/sous-commandes-cli-avec-sortie-d-aide-precise

Chapitre 181: Surcharge

Examples

Méthodes Magic / Dunder

Les méthodes magiques (également appelées dbr comme abréviation de double-trait de soulignement) en Python ont un objectif similaire à la surcharge d'opérateurs dans d'autres langages. Ils permettent à une classe de définir son comportement lorsqu'elle est utilisée comme opérande dans des expressions d'opérateur unaires ou binaires. Ils servent également d'implémentations appelées par certaines fonctions intégrées.

Considérons cette implémentation de vecteurs à deux dimensions.

```
import math

class Vector(object):
    # instantiation
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # unary negation (-v)
    def __neg__(self):
        return Vector(-self.x, -self.y)

    # addition (v + u)
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # subtraction (v - u)
    def __sub__(self, other):
        return self + (-other)

    # equality (v == u)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # abs(v)
    def __abs__(self):
        return math.hypot(self.x, self.y)

    # str(v)
    def __str__(self):
        return '<{0.x}, {0.y}>'.format(self)

    # repr(v)
    def __repr__(self):
        return 'Vector({0.x}, {0.y})'.format(self)
```

Il est maintenant possible d'utiliser naturellement des instances de la classe `Vector` dans diverses expressions.

```
v = Vector(1, 4)
u = Vector(2, 0)

u + v           # Vector(3, 4)
print(u + v)    # "<3, 4>" (implicit string conversion)
u - v           # Vector(1, -4)
u == v          # False
u + v == v + u  # True
abs(u + v)      # 5.0
```

Types de conteneur et de séquence

Il est possible d'émuler des types de conteneur, qui prennent en charge l'accès aux valeurs par clé ou par index.

Considérons cette implémentation naïve d'une liste fragmentée, qui stocke uniquement ses éléments non nuls pour conserver la mémoire.

```
class sparselist(object):
    def __init__(self, size):
        self.size = size
        self.data = {}

    # l[index]
    def __getitem__(self, index):
        if index < 0:
            index += self.size
        if index >= self.size:
            raise IndexError(index)
        try:
            return self.data[index]
        except KeyError:
            return 0.0

    # l[index] = value
    def __setitem__(self, index, value):
        self.data[index] = value

    # del l[index]
    def __delitem__(self, index):
        if index in self.data:
            del self.data[index]

    # value in l
    def __contains__(self, value):
        return value == 0.0 or value in self.data.values()

    # len(l)
    def __len__(self):
        return self.size

    # for value in l: ...
    def __iter__(self):
        return (self[i] for i in range(self.size)) # use xrange for python2
```

Ensuite, nous pouvons utiliser une liste `sparselist` comme une `list` régulière.

```
l = sparselist(10 ** 6)  # list with 1 million elements
0 in l                   # True
10 in l                  # False

l[12345] = 10
10 in l                  # True
l[12345]                 # 10

for v in l:
    pass  # 0, 0, 0, ... 10, 0, 0 ... 0
```

Types appelables

```
class adder(object):
    def __init__(self, first):
        self.first = first

    # a(...)
    def __call__(self, second):
        return self.first + second

add2 = adder(2)
add2(1)  # 3
add2(2)  # 4
```

Gestion du comportement non implémenté

Si votre classe n'implémente pas d'opérateur surchargé spécifique pour les types d'arguments fournis, elle doit `return NotImplemented` (**notez** qu'il s'agit d'une constante spéciale différente de `NotImplementedError`). Cela permettra à Python de recourir à d'autres méthodes pour faire fonctionner l'opération:

> Lorsque `NotImplemented` est renvoyé, l'interpréteur essaiera alors l'opération reflétée sur l'autre type, ou une autre solution de secours, selon l'opérateur. Si toutes les opérations tentées renvoient `NotImplemented` , l'interpréteur `NotImplemented` une exception appropriée.

Par exemple, avec `x + y` , si `x.__add__(y)` renvoie sans implémentation, `y.__radd__(x)` est tenté à la place.

```
class NotAddable(object):

    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return NotImplemented


class Addable(NotAddable):

    def __add__(self, other):
        return Addable(self.value + other.value)
```

```
    __radd__ = __add__
```

Comme c'est la méthode *réfléchie* , nous devons implémenter `__add__` **et** `__radd__` pour obtenir le comportement attendu dans tous les cas; Heureusement, comme ils font tous les deux la même chose dans cet exemple simple, nous pouvons prendre un raccourci.

Utilisé:

```
>>> x = NotAddable(1)
>>> y = Addable(2)
>>> x + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NotAddable' and 'NotAddable'
>>> y + y
<so.Addable object at 0x1095974d0>
>>> z = x + y
>>> z
<so.Addable object at 0x109597510>
>>> z.value
3
```

Surcharge de l'opérateur

Vous trouverez ci-dessous les opérateurs pouvant être surchargés dans les classes, ainsi que les définitions de méthode requises et un exemple de l'opérateur utilisé dans une expression.

NB L'utilisation d'un `other` nom de variable n'est pas obligatoire, mais est considérée comme la norme.

Opérateur	Méthode	Expression
`+` Ajout	`__add__(self, other)`	`a1 + a2`
`-` soustraction	`__sub__(self, other)`	`a1 - a2`
`*` Multiplication	`__mul__(self, other)`	`a1 * a2`
`@` Multiplication de matrices	`__matmul__(self, other)`	`a1 @ a2` (*Python 3.5*)
`/` Division	`__div__(self, other)`	`a1 / a2` (*Python 2 uniquement*)
`/` Division	`__truediv__(self, other)`	`a1 / a2` (*Python 3*)
`//` Division de plancher	`__floordiv__(self, other)`	`a1 // a2`
`%` Modulo / reste	`__mod__(self, other)`	`a1 % a2`
`**` puissance	`__pow__(self, other[, modulo])`	`a1 ** a2`
`<<`	`__lshift__(self, other)`	`a1 << a2`

Opérateur	Méthode	Expression
Changement bit à gauche		
`>>` Changement de bit à droite	`__rshift__(self, other)`	`a1 >> a2`
`&` Bitwise ET	`__and__(self, other)`	`a1 & a2`
`^` Bit-bit XOR	`__xor__(self, other)`	`a1 ^ a2`
`|` (Bit à bit OU)	`__or__(self, other)`	`a1 | a2`
`-` Négation (arithmétique)	`__neg__(self)`	`-a1`
`+` Positif	`__pos__(self)`	`+a1`
`~` Pas binaire	`__invert__(self)`	`~a1`
`<` Moins que	`__lt__(self, other)`	`a1 < a2`
`<=` Inférieur ou égal à	`__le__(self, other)`	`a1 <= a2`
`==` égal à	`__eq__(self, other)`	`a1 == a2`
`!=` Pas égal à	`__ne__(self, other)`	`a1 != a2`
`>` Supérieur à	`__gt__(self, other)`	`a1 > a2`
`>=` Supérieur ou égal à	`__ge__(self, other)`	`a1 >= a2`
`[index]` Opérateur d'index	`__getitem__(self, index)`	`a1[index]`
`in` opérateur In	`__contains__(self, other)`	`a2 in a1`
`(*args, ...)` Appel	`__call__(self, *args, **kwargs)`	`a1(*args, **kwargs)`

Le paramètre optionnel `modulo` pour `__pow__` n'est utilisé que par la fonction intégrée `pow` .

Chacune des méthodes correspondant à un opérateur *binaire* a une méthode "droite" correspondante qui commence par `__r` , par exemple `__radd__` :

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, other):
        return self.a + other
    def __radd__(self, other):
        print("radd")
        return other + self.a
```

```
A(1) + 2  # Out:  3
2 + A(1)  # prints radd. Out: 3
```

ainsi qu'une version correspondante en place, commençant par `__i` :

```
class B:
    def __init__(self, b):
        self.b = b
    def __iadd__(self, other):
        self.b += other
        print("iadd")
        return self

b = B(2)
b.b       # Out: 2
b += 1    # prints iadd
b.b       # Out: 3
```

Comme ces méthodes n'ont rien de particulier, de nombreuses autres parties du langage, des parties de la bibliothèque standard et même des modules tiers ajoutent des méthodes magiques, comme des méthodes pour convertir un objet en type ou vérifier les propriétés de l'objet. Par exemple, la fonction intégrée `str()` appelle la méthode `__str__` l'objet, si elle existe. Certaines de ces utilisations sont énumérées ci-dessous.

Fonction	Méthode	Expression
Casting à l' `int`	`__int__(self)`	`int(a1)`
Fonction absolue	`__abs__(self)`	`abs(a1)`
Casting à `str`	`__str__(self)`	`str(a1)`
Casting à `unicode`	`__unicode__(self)`	`unicode(a1)` (Python 2 uniquement)
Représentation de chaîne	`__repr__(self)`	`repr(a1)`
Casting à `bool`	`__nonzero__(self)`	`bool(a1)`
Formatage de chaîne	`__format__(self, formatstr)`	`"Hi {:abc}".format(a1)`
Hachage	`__hash__(self)`	`hash(a1)`
Longueur	`__len__(self)`	`len(a1)`
Renversé	`__reversed__(self)`	`reversed(a1)`
Sol	`__floor__(self)`	`math.floor(a1)`
Plafond	`__ceil__(self)`	`math.ceil(a1)`

Il existe également les méthodes spéciales `__enter__` et `__exit__` pour les gestionnaires de

contexte, et bien plus encore.

Lire Surcharge en ligne: https://riptutorial.com/fr/python/topic/2063/surcharge

Chapitre 182: sys

Introduction

Le module **sys** donne accès aux fonctions et aux valeurs concernant l'environnement d'exécution du programme, telles que les paramètres de ligne de commande dans `sys.argv` ou la fonction `sys.exit()` pour terminer le processus en cours à partir de n'importe quel point du flux de programme.

Bien que séparé dans un module, il est en fait intégré et, en tant que tel, sera toujours disponible dans des circonstances normales.

Syntaxe

- Importez le module sys et rendez-le disponible dans l'espace de noms actuel:

```
import sys
```

- Importez une fonction spécifique du module sys directement dans l'espace de noms actuel:

```
from sys import exit
```

Remarques

Pour plus de détails sur tous les membres du module **sys** , reportez-vous à la documentation officielle .

Examples

Arguments de ligne de commande

```
if len(sys.argv) != 4:         # The script name needs to be accounted for as well.
    raise RuntimeError("expected 3 command line arguments")

f = open(sys.argv[1], 'rb')    # Use first command line argument.
start_line = int(sys.argv[2])  # All arguments come as strings, so need to be
end_line = int(sys.argv[3])    # converted explicitly if other types are required.
```

Notez que dans les programmes plus grands et plus polis, vous utiliseriez des modules tels que click pour gérer les arguments de la ligne de commande au lieu de le faire vous-même.

Nom du script

```
# The name of the executed script is at the beginning of the argv list.
```

```
print('usage:', sys.argv[0], '<filename> <start> <end>')

# You can use it to generate the path prefix of the executed program
# (as opposed to the current module) to access files relative to that,
# which would be good for assets of a game, for instance.
program_file = sys.argv[0]

import pathlib
program_path = pathlib.Path(program_file).resolve().parent
```

Flux d'erreur standard

```
# Error messages should not go to standard output, if possible.
print('ERROR: We have no cheese at all.', file=sys.stderr)

try:
    f = open('nonexistent-file.xyz', 'rb')
except OSError as e:
    print(e, file=sys.stderr)
```

Terminer le processus prématurément et retourner un code de sortie

```
def main():
    if len(sys.argv) != 4 or '--help' in sys.argv[1:]:
        print('usage: my_program <arg1> <arg2> <arg3>', file=sys.stderr)

        sys.exit(1)    # use an exit code to signal the program was unsuccessful

    process_data()
```

Lire sys en ligne: https://riptutorial.com/fr/python/topic/9847/sys

Chapitre 183: Tableaux

Introduction

Les "tableaux" en Python ne sont pas les tableaux des langages de programmation classiques tels que C et Java, mais plus proches des listes. Une liste peut être une collection d'éléments homogènes ou hétérogènes et peut contenir des ints, des chaînes ou d'autres listes.

Paramètres

Paramètre	Détails
`b`	Représente un entier signé de taille 1 octet
`B`	Représente un entier non signé de taille 1 octet
`c`	Représente le caractère de taille 1 octet
`u`	Représente le caractère unicode de taille 2 octets
`h`	Représente un entier signé de taille 2 octets
`H`	Représente un entier non signé de taille 2 octets
`i`	Représente un entier signé de taille 2 octets
`I`	Représente un entier non signé de taille 2 octets
`w`	Représente le caractère unicode de taille 4 octets
`l`	Représente un entier signé de taille 4 octets
`L`	Représente un entier non signé de taille 4 octets
`f`	Représente un point flottant de taille 4 octets
`d`	Représente un point flottant de taille 8 octets

Examples

Introduction de base aux tableaux

Un tableau est une structure de données qui stocke les valeurs du même type de données. En Python, c'est la principale différence entre les tableaux et les listes.

Alors que les listes python peuvent contenir des valeurs correspondant à différents types de

données, les tableaux de python ne peuvent contenir que des valeurs correspondant au même type de données. Dans ce tutoriel, nous allons comprendre les tableaux Python avec quelques exemples.

Si vous êtes nouveau sur Python, commencez par l'article de Python Introduction.

Pour utiliser des tableaux en langage Python, vous devez importer le module de `array` standard. C'est parce que le tableau n'est pas un type de données fondamental comme les chaînes de caractères, les nombres entiers, etc. Voici comment importer `array` module de `array` dans python:

```
from array import *
```

Une fois le module de `array` importé, vous pouvez déclarer un tableau. Voici comment vous le faites:

```
arrayIdentifierName = array(typecode, [Initializers])
```

Dans la déclaration ci-dessus, `arrayIdentifierName` est le nom du tableau, `typecode` permet à python de connaître le type de tableau et les `Initializers` sont les valeurs avec lesquelles le tableau est initialisé.

Les types de code sont les codes utilisés pour définir le type des valeurs de tableau ou le type de tableau. Le tableau de la section des paramètres affiche les valeurs possibles que vous pouvez utiliser lors de la déclaration d'un tableau et de son type.

Voici un exemple concret de déclaration de tableau python:

```
my_array = array('i',[1,2,3,4])
```

Dans l'exemple ci-dessus, le type de code utilisé est `i` . Ce type de code représente un entier signé dont la taille est de 2 octets.

Voici un exemple simple d'un tableau contenant 5 entiers

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

Accéder à des éléments individuels via des index

Les éléments individuels sont accessibles via des index. Les tableaux Python sont indexés à zéro. Voici un exemple :

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
print(my_array[2])
# 3
print(my_array[0])
# 1
```

Ajoutez une valeur au tableau en utilisant la méthode append ()

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Notez que la valeur `6` été ajoutée aux valeurs de tableau existantes.

Insérer une valeur dans un tableau en utilisant la méthode insert ()

Nous pouvons utiliser la méthode `insert()` pour insérer une valeur à n'importe quel index du tableau. Voici un exemple :

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

Dans l'exemple ci-dessus, la valeur 0 a été insérée à l'index 0. Notez que le premier argument est l'index tandis que le second argument est la valeur.

Étendre le tableau python en utilisant la méthode extend ()

Un tableau python peut être étendu avec plusieurs valeurs à l'aide de la méthode `extend()` . Voici un exemple :

```
my_array = array('i', [1,2,3,4,5])
my_extnd_array = array('i', [7,8,9,10])
my_array.extend(my_extnd_array)
# array('i', [1, 2, 3, 4, 5, 7, 8, 9, 10])
```

Nous voyons que le tableau my_array a été étendu avec les valeurs de `my_extnd_array` .

Ajouter des éléments de la liste dans un tableau en utilisant la méthode fromlist ()

Voici un exemple:

```
my_array = array('i', [1,2,3,4,5])
c=[11,12,13]
my_array.fromlist(c)
# array('i', [1, 2, 3, 4, 5, 11, 12, 13])
```

Nous voyons donc que les valeurs 11,12 et 13 ont été ajoutées à partir de la liste `c` à `my_array` .

Supprimer tout élément de tableau en utilisant la méthode remove ()

Voici un exemple :

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

Nous voyons que l'élément 4 a été retiré du tableau.

Supprimer le dernier élément du tableau en utilisant la méthode pop ()

`pop` retire le dernier élément du tableau. Voici un exemple :

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

On voit donc que le dernier élément (`5`) est sorti du tableau.

Récupère n'importe quel élément via son index en utilisant la méthode index ()

`index()` renvoie le premier index de la valeur correspondante. Rappelez-vous que les tableaux sont indexés à zéro.

```
my_array = array('i', [1,2,3,4,5])
print(my_array.index(5))
# 5
my_array = array('i', [1,2,3,3,5])
print(my_array.index(3))
# 3
```

Notez dans ce deuxième exemple qu'un seul index a été renvoyé, même si la valeur existe deux fois dans le tableau

Inverser un tableau python en utilisant la méthode reverse ()

La méthode `reverse()` fait ce que le nom dit de faire - renverse le tableau. Voici un exemple :

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

Obtenir des informations sur les tampons de tableau via la méthode buffer_info ()

Cette méthode vous fournit l'adresse de début du tampon de tableau en mémoire et le nombre d'éléments dans le tableau. Voici un exemple:

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

Vérifier le nombre d'occurrences d'un élément en utilisant la méthode count ()

`count()` renverra le nombre de fois et l'élément apparaît dans un tableau. Dans l'exemple suivant, nous voyons que la valeur `3` apparaît deux fois.

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

Convertir un tableau en chaîne en utilisant la méthode tostring ()

`tostring()` convertit le tableau en chaîne.

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

Convertir un tableau en une liste python avec les mêmes éléments en utilisant la méthode tolist ()

Lorsque vous avez besoin d'un objet de `list` Python, vous pouvez utiliser la méthode `tolist()` pour convertir votre tableau en une liste.

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

Ajouter une chaîne au tableau de caractères en utilisant la méthode fromstring ()

Vous pouvez ajouter une chaîne à un tableau de caractères à l'aide de `fromstring()`

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

Lire Tableaux en ligne: https://riptutorial.com/fr/python/topic/4866/tableaux

Chapitre 184: Tableaux multidimensionnels

Examples

Listes dans les listes

Un bon moyen de visualiser un tableau 2D est une liste de listes. Quelque chose comme ça:

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

ici la liste extérieure `lst` a trois choses en elle. chacune de ces choses est une autre liste: la première est: `[1,2,3]` , la seconde est: `[4,5,6]` et la troisième est: `[7,8,9]` . Vous pouvez accéder à ces listes de la même manière que vous accéderiez à un autre élément d'une liste, comme ceci:

```
print (lst[0])
#output: [1, 2, 3]

print (lst[1])
#output: [4, 5, 6]

print (lst[2])
#output: [7, 8, 9]
```

Vous pouvez alors accéder aux différents éléments de chacune de ces listes de la même manière:

```
print (lst[0][0])
#output: 1

print (lst[0][1])
#output: 2
```

Ici, le premier chiffre entre crochets `[]` signifie que la liste se trouve dans cette position. Dans l'exemple ci-dessus, nous avons utilisé le nombre `0` pour obtenir la liste dans la position `0` qui est `[1,2,3]` . Le deuxième ensemble de `[]` signifie que l'objet est dans cette position à partir de la liste interne. Dans ce cas, nous avons utilisé à la fois `0` et `1` la 0ème position dans la liste que nous avons est le numéro `1` et dans la 1ère position, il est `2`

Vous pouvez également définir des valeurs dans ces listes de la même manière:

```
lst[0]=[10,11,12]
```

Maintenant, la liste est `[[10,11,12],[4,5,6],[7,8,9]]` . Dans cet exemple, nous avons changé toute la première liste pour en faire une liste complètement nouvelle.

```
lst[1][2]=15
```

Maintenant, la liste est `[[10,11,12],[4,5,15],[7,8,9]]` . Dans cet exemple, nous avons modifié un

seul élément à l'intérieur d'une des listes internes. D'abord, nous sommes entrés dans la liste à la position 1 et nous avons changé l'élément à la position 2, qui était 6 maintenant, c'est 15.

Listes dans les listes dans les listes dans ...

Ce comportement peut être étendu. Voici un tableau à 3 dimensions:

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,313]
```

Comme cela est probablement évident, cela devient un peu difficile à lire. Utilisez des barres obliques inversées pour diviser les différentes dimensions:

```
[[[111,112,113],[121,122,123],[131,132,133]],\
 [[211,212,213],[221,222,223],[231,232,233]],\
 [[311,312,313],[321,322,323],[331,332,333]]]
```

En imbriquant les listes comme ceci, vous pouvez étendre à des dimensions arbitrairement élevées.

L'accès est similaire aux tableaux 2D:

```
print(myarray)
print(myarray[1])
print(myarray[2][1])
print(myarray[1][0][2])
etc.
```

Et l'édition est également similaire:

```
myarray[1]=new_n-1_d_list
myarray[2][1]=new_n-2_d_list
myarray[1][0][2]=new_n-3_d_list #or a single number if you're dealing with 3D arrays
etc.
```

Lire Tableaux multidimensionnels en ligne: https://riptutorial.com/fr/python/topic/8186/tableaux-multidimensionnels

Chapitre 185: Tas

Examples

Les plus gros et les plus petits objets d'une collection

Pour trouver les plus gros éléments d'une collection, le module `heapq` a une fonction appelée `nlargest` , nous lui passons deux arguments, le premier est le nombre d'éléments à récupérer, le second est le nom de la collection:

```
import heapq


numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers))  # [200, 150, 100, 50]
```

De même, pour trouver les plus petits éléments d'une collection, nous utilisons la fonction `nsmallest` :

```
print(heapq.nsmallest(4, numbers))  # [1, 2, 4, 8]
```

Les fonctions `nlargest` et `nsmallest` un argument facultatif (paramètre clé) pour les structures de données complexes. L'exemple suivant montre l'utilisation de la propriété `age` pour récupérer le dictionnaire de `people` le plus ancien et le plus jeune:

```
people = [
    {'firstname': 'John', 'lastname': 'Doe', 'age': 30},
    {'firstname': 'Jane', 'lastname': 'Doe', 'age': 25},
    {'firstname': 'Janie', 'lastname': 'Doe', 'age': 10},
    {'firstname': 'Jane', 'lastname': 'Roe', 'age': 22},
    {'firstname': 'Johnny', 'lastname': 'Doe', 'age': 12},
    {'firstname': 'John', 'lastname': 'Roe', 'age': 45}
]

oldest = heapq.nlargest(2, people, key=lambda s: s['age'])
print(oldest)
# Output: [{'firstname': 'John', 'age': 45, 'lastname': 'Roe'}, {'firstname': 'John', 'age':
30, 'lastname': 'Doe'}]

youngest = heapq.nsmallest(2, people, key=lambda s: s['age'])
print(youngest)
# Output: [{'firstname': 'Janie', 'age': 10, 'lastname': 'Doe'}, {'firstname': 'Johnny',
'age': 12, 'lastname': 'Doe'}]
```

Le plus petit article d'une collection

La propriété la plus intéressante d'un `heap` est que son plus petit élément est toujours le premier élément: `heap[0]`

```
import heapq


numbers = [10, 4, 2, 100, 20, 50, 32, 200, 150, 8]

heapq.heapify(numbers)
print(numbers)
# Output: [2, 4, 10, 100, 8, 50, 32, 200, 150, 20]

heapq.heappop(numbers)  # 2
print(numbers)
# Output: [4, 8, 10, 100, 20, 50, 32, 200, 150]

heapq.heappop(numbers)  # 4
print(numbers)
# Output:  [8, 20, 10, 100, 150, 50, 32, 200]
```

Lire Tas en ligne: https://riptutorial.com/fr/python/topic/7489/tas

Chapitre 186: Test d'unité

Remarques

Il existe plusieurs outils de test unitaire pour Python. Cette rubrique de documentation décrit le module `unittest` base. Les autres outils de test incluent `py.test` et `nosetests` . Cette documentation python sur les tests compare plusieurs de ces outils sans entrer dans le détail.

Examples

Tester les exceptions

Les programmes lancent des erreurs lorsque, par exemple, des entrées incorrectes sont données. Pour cette raison, il faut s'assurer qu'une erreur est générée lorsque de fausses entrées sont fournies. Pour cette raison, nous devons vérifier l'existence d'une exception exacte. Dans cet exemple, nous utiliserons l'exception suivante:

```
class WrongInputException(Exception):
    pass
```

Cette exception est déclenchée lorsque des entrées incorrectes sont données, dans le contexte suivant, où nous attendons toujours un nombre comme entrée de texte.

```
def convert2number(random_input):
    try:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Expected an integer!")
    return my_input
```

Pour vérifier si une exception a été `assertRaises` , nous utilisons `assertRaises` pour vérifier cette exception. `assertRaises` peut être utilisé de deux manières:

1. Utilisation de l'appel de la fonction régulière. Le premier argument prend le type d'exception, le second un appelable (généralement une fonction) et le reste des arguments est transmis à cet appelable.
2. En utilisant une clause `with` , en ne donnant que le type d'exception à la fonction. Cela a pour avantage que plus de code peut être exécuté, mais doit être utilisé avec précaution car plusieurs fonctions peuvent utiliser la même exception, ce qui peut être problématique. Un exemple: avec self.assertRaises (WrongInputException): convert2number ("pas un nombre")

Ce premier a été implémenté dans le cas de test suivant:

```
import unittest

class ExceptionTestCase(unittest.TestCase):
```

```
    def test_wrong_input_string(self):
        self.assertRaises(WrongInputException, convert2number, "not a number")

    def test_correct_input(self):
        try:
            result = convert2number("56")
            self.assertIsInstance(result, int)
        except WrongInputException:
            self.fail()
```

Il peut également être nécessaire de vérifier une exception qui n'aurait pas dû être lancée. Cependant, un test échouera automatiquement lorsqu'une exception est lancée et peut ne pas être nécessaire du tout. Juste pour montrer les options, la seconde méthode de test montre comment vérifier qu'une exception ne peut pas être lancée. Fondamentalement, cela intercepte l'exception et échoue ensuite le test en utilisant la méthode `fail` .

Fonctions moqueuses avec unittest.mock.create_autospec

Une façon de simuler une fonction consiste à utiliser la fonction `create_autospec` , qui va `create_autospec` un objet en fonction de ses spécifications. Avec les fonctions, nous pouvons l'utiliser pour nous assurer qu'elles sont correctement appelées.

Avec une fonction `multiply` dans `custom_math.py` :

```
def multiply(a, b):
    return a * b
```

Et une fonction `multiples_of` dans `process_math.py` :

```
from custom_math import multiply


def multiples_of(integer, *args, num_multiples=0, **kwargs):
    """
    :rtype: list
    """
    multiples = []

    for x in range(1, num_multiples + 1):
        """
        Passing in args and kwargs here will only raise TypeError if values were
        passed to multiples_of function, otherwise they are ignored. This way we can
        test that multiples_of is used correctly. This is here for an illustration
        of how create_autospec works. Not recommended for production code.
        """
        multiple = multiply(integer,x, *args, **kwargs)
        multiples.append(multiple)

    return multiples
```

On peut tester les `multiples_of` seul en se moquant de la `multiply` . L'exemple ci-dessous utilise la bibliothèque standard Python unittest, mais elle peut également être utilisée avec d'autres frameworks de test, comme pytest ou nose:

```
from unittest.mock import create_autospec
import unittest

# we import the entire module so we can mock out multiply
import custom_math
custom_math.multiply = create_autospec(custom_math.multiply)
from process_math import multiples_of


class TestCustomMath(unittest.TestCase):
    def test_multiples_of(self):
        multiples = multiples_of(3, num_multiples=1)
        custom_math.multiply.assert_called_with(3, 1)

    def test_multiples_of_with_bad_inputs(self):
        with self.assertRaises(TypeError) as e:
            multiples_of(1, "extra arg",  num_multiples=1) # this should raise a TypeError
```

Tester la configuration et le démontage dans un fichier unestest.TestCase

Parfois, nous voulons préparer un contexte pour chaque test à exécuter. La méthode `setUp` est exécutée avant chaque test de la classe. `tearDown` est exécuté à la fin de chaque test. Ces méthodes sont facultatives. Rappelez-vous que les TestCases sont souvent utilisés dans l'héritage multiple coopératif, vous devez donc toujours faire appel à `super` dans ces méthodes pour que les méthodes `setUp` et `tearDown` la classe de base `setUp` également appelées. L'implémentation de base de `TestCase` fournit des méthodes `setUp` et `tearDown` vides afin qu'elles puissent être appelées sans générer d'exceptions:

```
import unittest


class SomeTest(unittest.TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.mock_data = [1,2,3,4,5]

    def test(self):
        self.assertEqual(len(self.mock_data), 5)

    def tearDown(self):
        super(SomeTest, self).tearDown()
        self.mock_data = []


if __name__ == '__main__':
    unittest.main()
```

Notez que dans python2.7 +, il existe également la méthode `addCleanup` qui enregistre les fonctions à appeler après l'exécution du test. Contrairement à `tearDown` qui n'est appelé que si `setUp` réussit, les fonctions enregistrées via `addCleanup` seront appelées même en cas d'exception non `setUp` dans `setUp` . À titre d'exemple concret, cette méthode peut souvent être vue en train de supprimer divers objets enregistrés lors de l'exécution du test:

```
import unittest
```

```
import some_module


class SomeOtherTest(unittest.TestCase):
    def setUp(self):
        super(SomeOtherTest, self).setUp()

        # Replace `some_module.method` with a `mock.Mock`
        my_patch = mock.patch.object(some_module, 'method')
        my_patch.start()

        # When the test finishes running, put the original method back.
        self.addCleanup(my_patch.stop)
```

Un autre avantage de l'enregistrement des nettoyages de cette manière est qu'il permet au programmeur de placer le code de nettoyage à côté du code de configuration et qu'il vous protège si un sous-programme oublie d'appeler `super` in `tearDown` .

Affirmer des exceptions

Vous pouvez tester qu'une fonction émet une exception avec le Unittest intégré via deux méthodes différentes.

Utiliser un gestionnaire de contexte

```
def division_function(dividend, divisor):
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError):
            x = division_function(1, 0)
```

Cela exécutera le code à l'intérieur du gestionnaire de contexte et, s'il réussit, le test échouera car l'exception n'a pas été déclenchée. Si le code déclenche une exception du type correct, le test continuera.

Vous pouvez également obtenir le contenu de l'exception déclenchée si vous souhaitez exécuter des assertions supplémentaires.

```
class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError) as ex:
            x = division_function(1, 0)

        self.assertEqual(ex.message, 'integer division or modulo by zero')
```

En fournissant une fonction appelable

```
def division_function(dividend, divisor):
    """
    Dividing two numbers.
```

```
    :type dividend: int
    :type divisor: int

    :raises: ZeroDivisionError if divisor is zero (0).
    :rtype: int
    """
    return dividend / divisor


class MyTestCase(unittest.TestCase):
    def test_passing_function(self):
        self.assertRaises(ZeroDivisionError, division_function, 1, 0)
```

L'exception à vérifier doit être le premier paramètre et une fonction appelable doit être transmise en tant que second paramètre. Tous les autres paramètres spécifiés seront transmis directement à la fonction appelée, vous permettant de spécifier les paramètres qui déclenchent l'exception.

Choisir des assertions au sein des inattaquables

Bien que Python ait une déclaration `assert` , l'infrastructure de test unitaire Python possède de meilleures assertions spécialisées pour les tests: ils sont plus informatifs sur les échecs et ne dépendent pas du mode de débogage de l'exécution.

L'assertion la plus simple est peut-être `assertTrue` , qui peut être utilisée comme ceci:

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test_basic(self):
        self.assertTrue(1 + 1 == 2)
```

Cela se passera bien, mais en remplaçant la ligne ci-dessus par

```
        self.assertTrue(1 + 1 == 3)
```

va échouer.

L'assertion `assertTrue` est probablement l'assertion la plus générale, car tout élément testé peut être considéré comme une condition booléenne, mais il existe souvent de meilleures alternatives. En testant l'égalité, comme ci-dessus, il est préférable d'écrire

```
        self.assertEqual(1 + 1, 3)
```

Lorsque le premier échoue, le message est

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------
```

```
Traceback (most recent call last):

  File "stuff.py", line 6, in test

    self.assertTrue(1 + 1 == 3)

AssertionError: False is not true
```

mais lorsque ce dernier échoue, le message est

```
======================================================================

FAIL: test (__main__.TruthTest)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "stuff.py", line 6, in test

    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
```

ce qui est plus informatif (il a en fait évalué le résultat du côté gauche).

Vous pouvez trouver la liste des assertions dans la documentation standard . En général, il est judicieux de choisir l'assertion la plus adaptée à la situation. Ainsi, comme indiqué ci-dessus, pour affirmer que `1 + 1 == 2` il est préférable d'utiliser `assertEqual` que `assertTrue` . De même, pour affirmer que `a is None` , il est préférable d'utiliser `assertIsNone` que `assertEqual` .

Notez également que les assertions ont des formes négatives. Ainsi `assertEqual` a son homologue négatif `assertNotEqual` et `assertIsNone` a son homologue négatif `assertIsNotNone` . Encore une fois, en utilisant les contreparties négatives, le cas échéant, cela conduira à des messages d'erreur plus clairs.

Tests unitaires avec le pytest

installation de pytest:

```
pip install pytest
```

préparer les tests:

```
mkdir tests
touch tests/test_docker.py
```

Fonctions à tester dans `docker_something/helpers.py` :

```
from subprocess import Popen, PIPE
# this Popen is monkeypatched with the fixture `all_popens`

def copy_file_to_docker(src, dest):
```

```
    try:
        result = Popen(['docker','cp', src, 'something_cont:{}'.format(dest)], stdout=PIPE,
stderr=PIPE)
        err = result.stderr.read()
        if err:
            raise Exception(err)
    except Exception as e:
        print(e)
    return result

def docker_exec_something(something_file_string):
    fl = Popen(["docker", "exec", "-i", "something_cont", "something"], stdin=PIPE,
stdout=PIPE, stderr=PIPE)
    fl.stdin.write(something_file_string)
    fl.stdin.close()
    err = fl.stderr.read()
    fl.stderr.close()
    if err:
        print(err)
        exit()
    result = fl.stdout.read()
    print(result)
```

Le test importe `test_docker.py` :

```
import os
from tempfile import NamedTemporaryFile
import pytest
from subprocess import Popen, PIPE

from docker_something import helpers
copy_file_to_docker = helpers.copy_file_to_docker
docker_exec_something = helpers.docker_exec_something
```

se moquer d'un fichier comme objet dans `test_docker.py` :

```
class MockBytes():
    '''Used to collect bytes
    '''
    all_read = []
    all_write = []
    all_close = []

    def read(self, *args, **kwargs):
        # print('read', args, kwargs, dir(self))
        self.all_read.append((self, args, kwargs))

    def write(self, *args, **kwargs):
        # print('wrote', args, kwargs)
        self.all_write.append((self, args, kwargs))

    def close(self, *args, **kwargs):
        # print('closed', self, args, kwargs)
        self.all_close.append((self, args, kwargs))

    def get_all_mock_bytes(self):
        return self.all_read, self.all_write, self.all_close
```

Patch de singe avec pytest dans `test_docker.py` :

```
@pytest.fixture
def all_popens(monkeypatch):
    '''This fixture overrides / mocks the builtin Popen
        and replaces stdin, stdout, stderr with a MockBytes object

        note: monkeypatch is magically imported
    '''
    all_popens = []

    class MockPopen(object):
        def __init__(self, args, stdout=None, stdin=None, stderr=None):
            all_popens.append(self)
            self.args = args
            self.byte_collection = MockBytes()
            self.stdin = self.byte_collection
            self.stdout = self.byte_collection
            self.stderr = self.byte_collection
            pass
    monkeypatch.setattr(helpers, 'Popen', MockPopen)

    return all_popens
```

Les exemples de tests doivent commencer par le préfixe `test_` dans le fichier `test_docker.py` :

```
def test_docker_install():
    p = Popen(['which', 'docker'], stdout=PIPE, stderr=PIPE)
    result = p.stdout.read()
    assert 'bin/docker' in result

def test_copy_file_to_docker(all_popens):
    result = copy_file_to_docker('asdf', 'asdf')
    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert mock_read
    assert result.args == ['docker', 'cp', 'asdf', 'something_cont:asdf']


def test_docker_exec_something(all_popens):

    docker_exec_something(something_file_string)

    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert len(mock_read) == 3
    something_template_stdin = mock_write[0][1][0]
    these = [os.environ['USER'], os.environ['password_prod'], 'table_name_here', 'test_vdm',
'col_a', 'col_b', '/tmp/test.tsv']
    assert all([x in something_template_stdin for x in these])
```

exécuter les tests un par un:

```
py.test -k test_docker_install tests
py.test -k test_copy_file_to_docker tests
py.test -k test_docker_exec_something tests
```

exécuter tous les tests dans le dossier de `tests` :

```
py.test -k test_ tests
```

Lire Test d'unité en ligne: https://riptutorial.com/fr/python/topic/631/test-d-unite

Chapitre 187: tkinter

Introduction

Sorti dans Tkinter, il est la bibliothèque d'interfaces graphiques la plus populaire de Python. Cette rubrique explique l'utilisation correcte de cette bibliothèque et de ses fonctionnalités.

Remarques

La capitalisation du module tkinter est différente entre Python 2 et 3. Pour Python 2, utilisez ce qui suit:

```
from Tkinter import *  # Capitalized
```

Pour Python 3, utilisez ce qui suit:

```
from tkinter import *  # Lowercase
```

Pour le code qui fonctionne avec Python 2 et 3, vous pouvez soit faire

```
try:
    from Tkinter import *
except ImportError:
    from tkinter import *
```

ou

```
from sys import version_info
if version_info.major == 2:
    from Tkinter import *
elif version_info.major == 3:
    from tkinter import *
```

> Voir la documentation de tkinter pour plus de détails

Examples

Une application tkinter minimale

`tkinter` est une boîte à outils graphique qui fournit un wrapper autour de la bibliothèque d'interface graphique Tk / Tcl et est incluse avec Python. Le code suivant crée une nouvelle fenêtre à l'aide de `tkinter` et place du texte dans le corps de la fenêtre.

> Remarque: Dans Python 2, la capitalisation peut être légèrement différente, voir la section Remarques ci-dessous.

```
import tkinter as tk

# GUI window is a subclass of the basic tkinter Frame object
class HelloWorldFrame(tk.Frame):
    def __init__(self, master):
        # Call superclass constructor
        tk.Frame.__init__(self, master)
        # Place frame into main window
        self.grid()
        # Create text box with "Hello World" text
        hello = tk.Label(self, text="Hello World! This label can hold strings!")
        # Place text box into frame
        hello.grid(row=0, column=0)

# Spawn window
if __name__ == "__main__":
    # Create main window object
    root = tk.Tk()
    # Set title of window
    root.title("Hello World!")
    # Instantiate HelloWorldFrame object
    hello_frame = HelloWorldFrame(root)
    # Start GUI
    hello_frame.mainloop()
```

Gestionnaires de géométrie

Tkinter dispose de trois mécanismes de gestion de la géométrie: `place` , `pack` et `grid` .

Le gestionnaire de `place` utilise des coordonnées de pixels absolues.

Le gestionnaire de `pack` place les widgets sur l'un des 4 côtés. Les nouveaux widgets sont placés à côté des widgets existants.

Le gestionnaire de `grid` place les widgets dans une grille similaire à une feuille de calcul à redimensionnement dynamique.

Endroit

Les arguments les plus courants pour `widget.place` sont les suivants:

- `x` , la coordonnée x absolue du widget
- `y` , coordonnée absolue du widget
- `height` , la hauteur absolue du widget
- `width` , la largeur absolue du widget

Un exemple de code utilisant `place` :

```
class PlaceExample(Frame):
    def __init__(self,master):
        Frame.__init__(self,master)
        self.grid()
        top_text=Label(master,text="This is on top at the origin")
```

```
        #top_text.pack()
        top_text.place(x=0,y=0,height=50,width=200)
        bottom_right_text=Label(master,text="This is at position 200,400")
        #top_text.pack()
        bottom_right_text.place(x=200,y=400,height=50,width=200)
# Spawn Window
if __name__=="__main__":
    root=Tk()
    place_frame=PlaceExample(root)
    place_frame.mainloop()
```

Pack

`widget.pack` peut prendre les arguments suivants:

- `expand` , remplir ou non l'espace laissé par le parent
- `fill` , s'il faut développer pour remplir tout l'espace (NONE (par défaut), X, Y ou BOTH)
- `side` , le côté à emballer contre (TOP (par défaut), BOTTOM, LEFT ou RIGHT)

la grille

Les arguments de mots-clés les plus utilisés de `widget.grid` sont les suivants:

- `row` , la ligne du widget (la plus petite valeur par défaut inoccupée)
- `rowspan` , le nombre de colonnes d'un widget (par défaut 1)
- `column` , la colonne du widget (0 par défaut)
- `columnspan` , le nombre de colonnes d'un widget (par défaut 1)
- `sticky` , où placer le widget si la cellule de la grille est plus grande que cela (combinaison de N, NE, E, SE, S, SW, W, NW)

Les lignes et les colonnes sont indexées à zéro. Les rangs augmentent en baisse et les colonnes augmentent à droite.

Un exemple de code utilisant la `grid` :

```
from tkinter import *

class GridExample(Frame):
    def __init__(self,master):
        Frame.__init__(self,master)
        self.grid()
        top_text=Label(self,text="This text appears on top left")
        top_text.grid() # Default position 0, 0
        bottom_text=Label(self,text="This text appears on bottom left")
        bottom_text.grid() # Default position 1, 0
        right_text=Label(self,text="This text appears on the right and spans both rows",
                         wraplength=100)
        # Position is 0,1
        # Rowspan means actual position is [0-1],1
        right_text.grid(row=0,column=1,rowspan=2)
```

```
# Spawn Window
if __name__=="__main__":
    root=Tk()
    grid_frame=GridExample(root)
    grid_frame.mainloop()
```

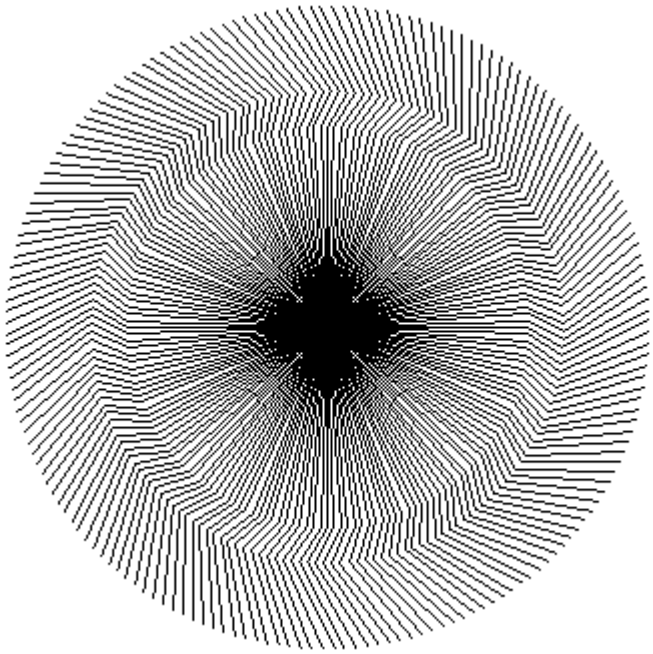
Ne mélangez jamais le `pack` et la `grid` dans le même cadre! Cela conduirait à une impasse de l'application!

Lire tkinter en ligne: https://riptutorial.com/fr/python/topic/7574/tkinter

Chapitre 188: Tortue Graphiques

Examples

Ninja Twist (Graphiques Tortue)



Voici une tortue graphique Ninja Twist:

```
import turtle

ninja = turtle.Turtle()

ninja.speed(10)

for i in range(180):
    ninja.forward(100)
    ninja.right(30)
    ninja.forward(20)
    ninja.left(60)
    ninja.forward(50)
    ninja.right(30)

    ninja.penup()
    ninja.setposition(0, 0)
    ninja.pendown()

    ninja.right(2)

turtle.done()
```

Lire Tortue Graphiques en ligne: https://riptutorial.com/fr/python/topic/7915/tortue-graphiques

Chapitre 189: Tracer avec Matplotlib

Introduction

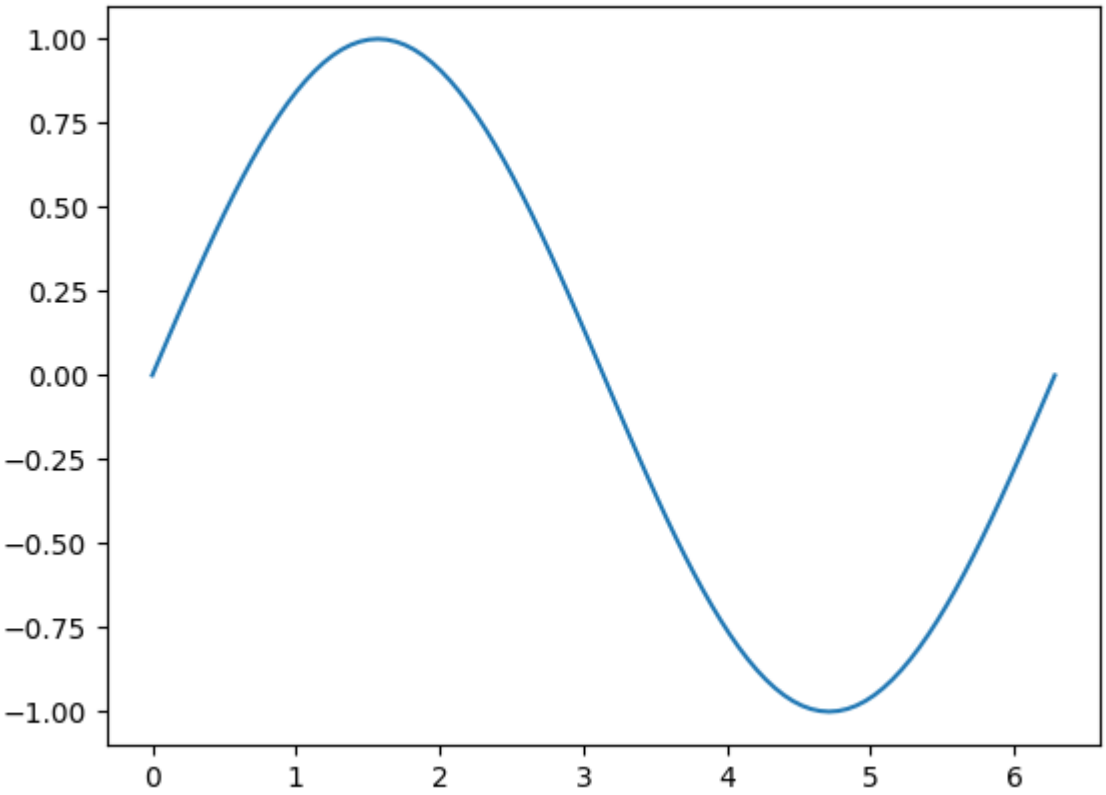
Matplotlib (https://matplotlib.org/) est une bibliothèque de traçage 2D basée sur NumPy. Voici quelques exemples de base. Vous trouverez d'autres exemples dans la documentation officielle (https://matplotlib.org/2.0.2/gallery.html et https://matplotlib.org/2.0.2/examples/index.html) ainsi que dans http: //www.riptutorial.com/topic/881

Examples

Une parcelle simple dans Matplotlib

Cet exemple illustre comment créer une courbe sinusoïdale simple à l'aide de **Matplotlib**

```
# Plotting tutorials in Python
# Launching a simple plot

import numpy as np
import matplotlib.pyplot as plt

# angle varying between 0 and 2*pi
x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)                         # sine function

plt.plot(x, y)
plt.show()
```

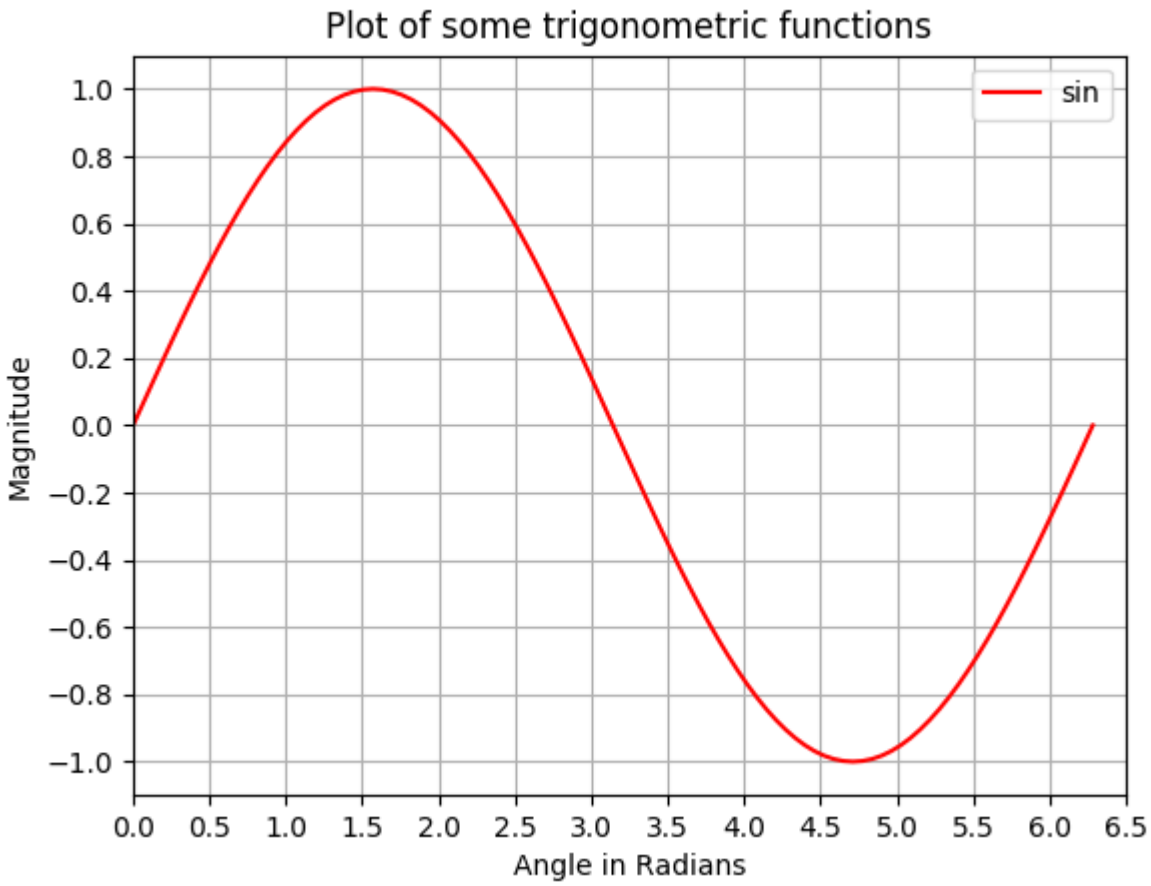


Ajout de plusieurs fonctionnalités à un tracé simple: libellés d'axe, titre, ticks d'axe, grille et légende

Dans cet exemple, nous prenons une courbe sinusoïdale et y ajoutons plus de fonctionnalités; à savoir le titre, les étiquettes des axes, le titre, les graduations des axes, la grille et la légende.

```
# Plotting tutorials in Python
# Enhancing a plot

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
```

```
plt.show()
```

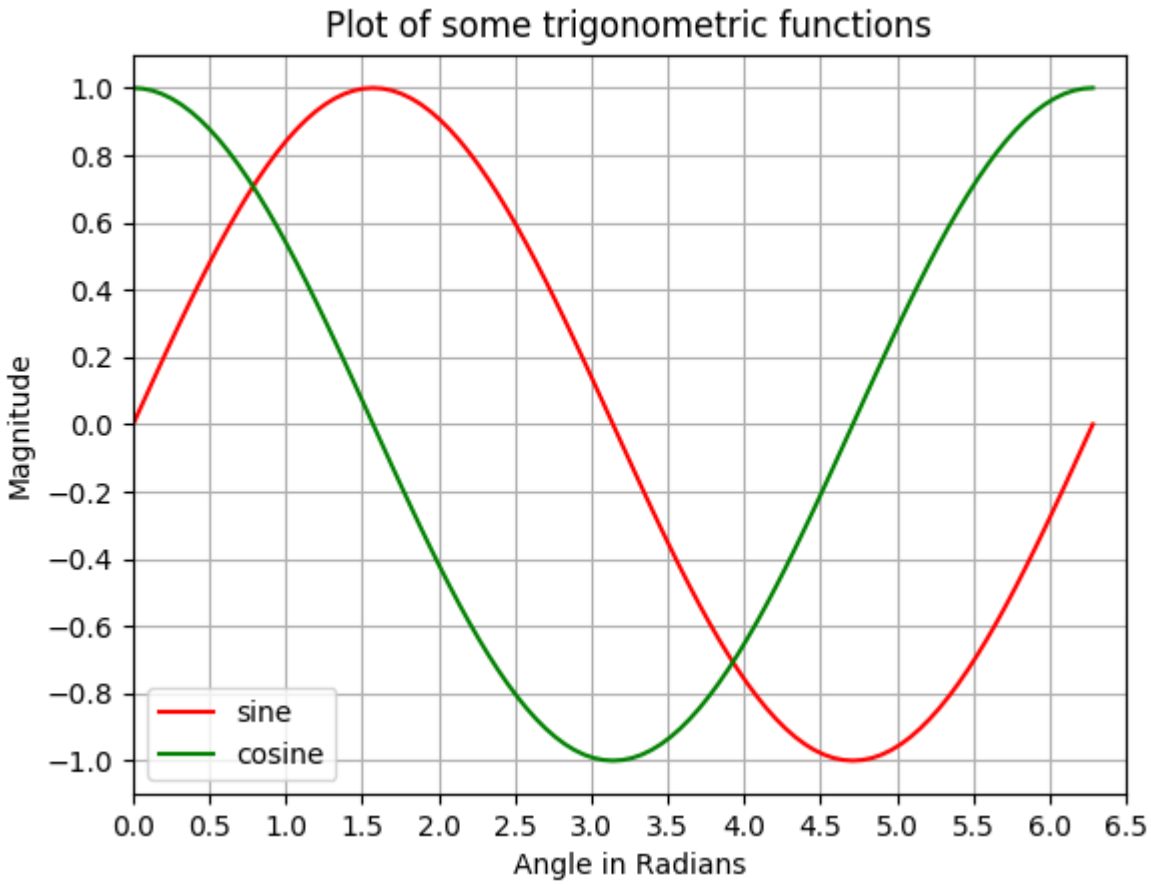


Faire plusieurs tracés dans la même figure par superposition similaire à MATLAB

Dans cet exemple, une courbe sinusoïdale et une courbe en cosinus sont tracées sur la même figure en superposant les tracés les uns sur les autres.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using single plot command and legend

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, 'r', x, z, 'g') # r, g - red, green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
```

```
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend(['sine', 'cosine'])
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```

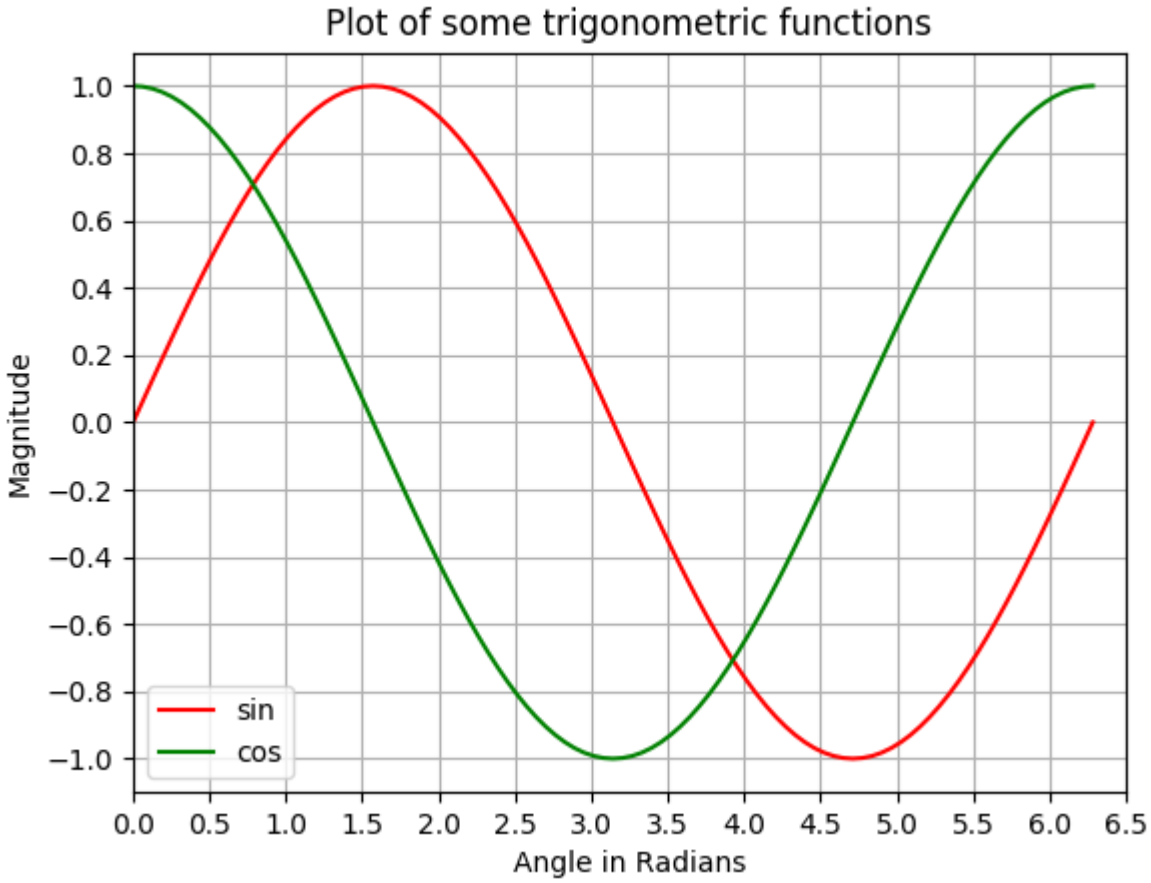


Réaliser plusieurs tracés dans la même figure en utilisant la superposition de tracé avec des commandes de tracé séparées

Comme dans l'exemple précédent, une courbe sinus et une courbe cosinus sont représentées sur la même figure à l'aide de commandes de tracé séparées. Ceci est plus pythonique et peut être utilisé pour obtenir des poignées séparées pour chaque tracé.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using multiple plot commands
# Much better and preferred than previous

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)
```

```
# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.plot(x, z, color='g', label='cos') # g - green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```

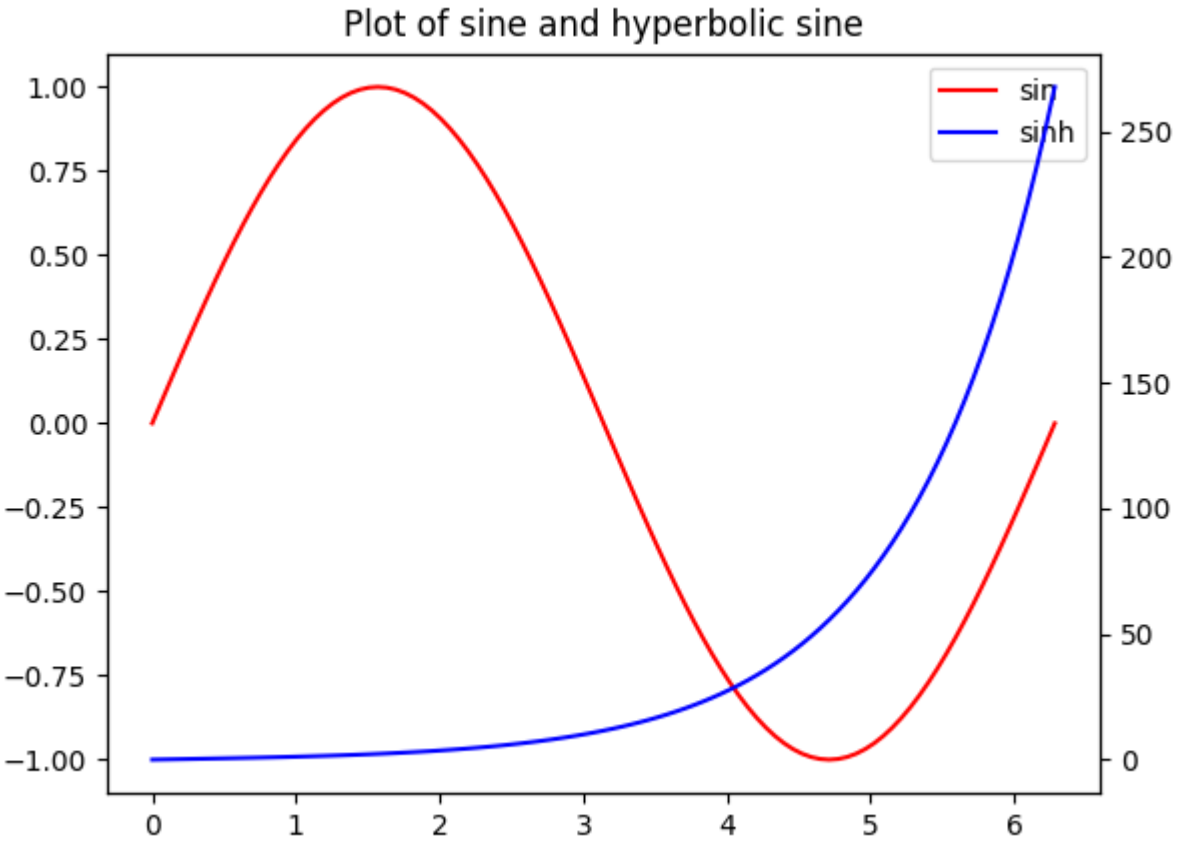


Tracés avec axe X commun mais axe Y différent: Utilisation de twinx ()

Dans cet exemple, nous allons tracer une courbe sinusoïdale et une courbe sinusoïdale hyperbolique dans le même tracé avec un axe X commun ayant un axe y différent. Ceci est accompli en utilisant la commande **twinx ()** .

```
# Plotting tutorials in Python
# Adding Multiple plots by twin x axis
# Good for plots having different y axis range
# Separate axes and figure objects
```

```
# replicate axes object and plot curves
# use axes to set attributes

# Note:
# Grid for second curve unsuccessful : let me know if you find it! :(

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.sinh(x)

# separate the figure object and axes object
# from the plotting object
fig, ax1 = plt.subplots()

# Duplicate the axes with a different y axis
# and the same x axis
ax2 = ax1.twinx() # ax2 and ax1 will have common x axis and different y axis

# plot the curves on axes 1, and 2, and get the curve handles
curve1, = ax1.plot(x, y, label="sin", color='r')
curve2, = ax2.plot(x, z, label="sinh", color='b')

# Make a curves list to access the parameters in the curves
curves = [curve1, curve2]

# add legend via axes 1 or axes 2 object.
# one command is usually sufficient
# ax1.legend() # will not display the legend of ax2
# ax2.legend() # will not display the legend of ax1
ax1.legend(curves, [curve.get_label() for curve in curves])
# ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid

# Global figure properties
plt.title("Plot of sine and hyperbolic sine")
plt.show()
```

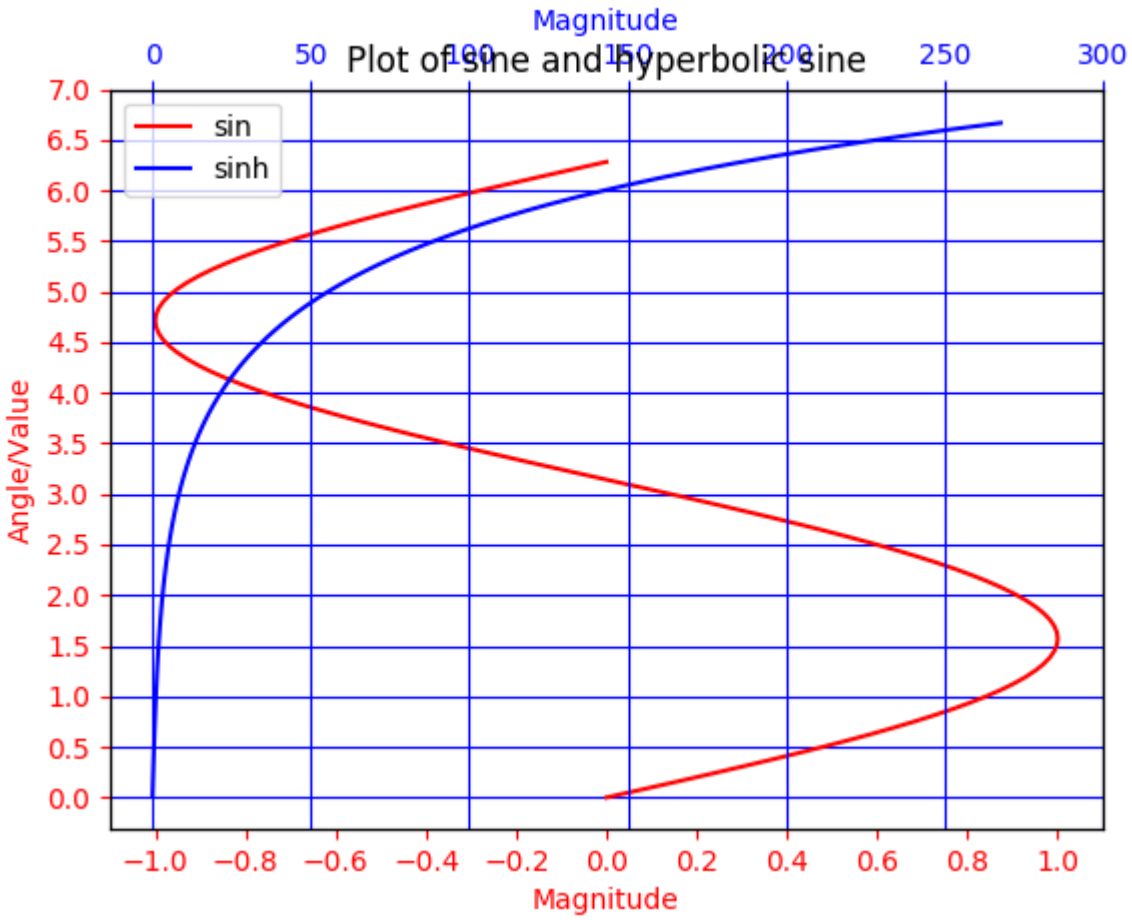


Tracés avec axe Y commun et axe X différent utilisant twiny ()

Dans cet exemple, un tracé avec des courbes ayant un axe y commun mais un axe des abscisses différent est démontré à l'aide de la méthode **twiny ()** . En outre, certaines fonctionnalités supplémentaires telles que le titre, la légende, les étiquettes, les grilles, les graduations des axes et les couleurs sont ajoutées au tracé.

```
# Plotting tutorials in Python
# Adding Multiple plots by twin y axis
# Good for plots having different x axis range
# Separate axes and figure objects
# replicate axes object and plot curves
# use axes to set attributes

import numpy as np
import matplotlib.pyplot as plt

y = np.linspace(0, 2.0*np.pi, 101)
x1 = np.sin(y)
x2 = np.sinh(y)

# values for making ticks in x and y axis
ynumbers = np.linspace(0, 7, 15)
xnumbers1 = np.linspace(-1, 1, 11)
xnumbers2 = np.linspace(0, 300, 7)

# separate the figure object and axes object
# from the plotting object
```

```
fig, ax1 = plt.subplots()

# Duplicate the axes with a different x axis
# and the same y axis
ax2 = ax1.twiny() # ax2 and ax1 will have common y axis and different x axis

# plot the curves on axes 1, and 2, and get the axes handles
curve1, = ax1.plot(x1, y, label="sin", color='r')
curve2, = ax2.plot(x2, y, label="sinh", color='b')

# Make a curves list to access the parameters in the curves
curves = [curve1, curve2]

# add legend via axes 1 or axes 2 object.
# one command is usually sufficient
# ax1.legend() # will not display the legend of ax2
# ax2.legend() # will not display the legend of ax1
# ax1.legend(curves, [curve.get_label() for curve in curves])
ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid

# x axis labels via the axes
ax1.set_xlabel("Magnitude", color=curve1.get_color())
ax2.set_xlabel("Magnitude", color=curve2.get_color())

# y axis label via the axes
ax1.set_ylabel("Angle/Value", color=curve1.get_color())
# ax2.set_ylabel("Magnitude", color=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# y ticks - make them coloured as well
ax1.tick_params(axis='y', colors=curve1.get_color())
# ax2.tick_params(axis='y', colors=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# x axis ticks via the axes
ax1.tick_params(axis='x', colors=curve1.get_color())
ax2.tick_params(axis='x', colors=curve2.get_color())

# set x ticks
ax1.set_xticks(xnumbers1)
ax2.set_xticks(xnumbers2)

# set y ticks
ax1.set_yticks(ynumbers)
# ax2.set_yticks(ynumbers) # also works

# Grids via axes 1 # use this if axes 1 is used to
# define the properties of common x axis
# ax1.grid(color=curve1.get_color())

# To make grids using axes 2
ax1.grid(color=curve2.get_color())
ax2.grid(color=curve2.get_color())
ax1.xaxis.grid(False)

# Global figure properties
plt.title("Plot of sine and hyperbolic sine")
plt.show()
```



Lire Tracer avec Matplotlib en ligne: https://riptutorial.com/fr/python/topic/10264/tracer-avec-matplotlib

Chapitre 190: Travailler autour du verrou d'interprète global (GIL)

Remarques

Pourquoi y a-t-il un GIL?

Le GIL existe dans CPython depuis la création des threads Python, en 1992. Il est conçu pour garantir la sécurité des threads lors de l'exécution du code python. Les interpréteurs Python écrits avec un GIL empêchent plusieurs threads natifs d'exécuter les bytecodes Python à la fois. Cela permet aux plug-ins de s'assurer que leur code est thread-safe: verrouillez simplement le GIL et seul votre thread actif peut s'exécuter, votre code est donc automatiquement compatible avec les threads.

Version courte: le GIL garantit que peu importe le nombre de processeurs et de threads que vous avez, *un seul thread d'un interpréteur Python s'exécutera en même temps.*

Cela présente de nombreux avantages en termes de facilité d'utilisation, mais présente également de nombreux avantages négatifs.

Notez qu'un GIL n'est pas une exigence du langage Python. Par conséquent, vous ne pouvez pas accéder au GIL directement à partir du code python standard. Toutes les implémentations de Python n'utilisent pas de GIL.

Interprètes ayant un GIL: CPython, PyPy, Cython (mais vous pouvez désactiver le GIL avec `nogil`)

Interprètes sans GIL: Jython, IronPython

Détails sur le fonctionnement du GIL:

Lorsqu'un thread est en cours d'exécution, il verrouille le GIL. Lorsqu'un thread veut s'exécuter, il demande le GIL et attend qu'il soit disponible. Dans CPython, avant la version 3.2, le thread en cours d'exécution vérifiait après un certain nombre d'instructions python pour voir si un autre code voulait le verrou (c'est-à-dire qu'il libérait le verrou et le demandait à nouveau). Cette méthode avait tendance à provoquer la famine des threads, en grande partie parce que le thread qui libérait le verrou l'acquitterait à nouveau avant que les threads en attente aient une chance de se réveiller. Depuis 3.2, les threads qui veulent le GIL attendent le verrouillage pendant un certain temps et, après ce temps, ils définissent une variable partagée qui force le thread en cours d'exécution à céder. Cela peut néanmoins entraîner des délais d'exécution considérablement plus longs. Voir les liens ci-dessous à partir de dabeaz.com (dans la section des références) pour plus de détails.

CPython libère automatiquement le GIL lorsqu'un thread effectue une opération d'E / S. Les bibliothèques de traitement d'images et les opérations de calcul numérique numérotées libèrent le GIL avant d'effectuer leur traitement.

Avantages du GIL

Pour les interprètes qui utilisent le GIL, le GIL est systémique. Il est utilisé pour préserver l'état de l'application. Les avantages comprennent:

- Collecte des ordures - Le nombre de références thread-safe doit être modifié lorsque le GIL est verrouillé. *Dans CPython, toute la collection Garbarge est liée au GIL.* C'est un gros voir l'article wiki python.org sur le GIL (listé dans Références ci-dessous) pour plus de détails sur ce qui doit encore être fonctionnel si l'on voulait supprimer le GIL.
- Facilité pour les programmeurs traitant du GIL - le verrouillage de tout est simpliste, mais facile à coder
- Facilite l'importation de modules d'autres langages

Conséquences du GIL

Le GIL n'autorise qu'un seul thread à exécuter du code python à la fois dans l'interpréteur python. Cela signifie que le multithreading de processus exécutant un code Python strict ne fonctionne tout simplement pas. Lorsque vous utilisez des threads sur le GIL, vous aurez probablement de moins bonnes performances avec les threads que si vous les exécutiez dans un seul thread.

Les références:

https://wiki.python.org/moin/GlobalInterpreterLock - résumé rapide de ce qu'il fait, des détails précis sur tous les avantages

http://programmers.stackexchange.com/questions/186889/why-was-python-written-with-the-gil - résumé clairement écrit

http://www.dabeaz.com/python/UnderstandingGIL.pdf - comment fonctionne le GIL et pourquoi il ralentit sur plusieurs cœurs

http://www.dabeaz.com/GIL/gilvis/index.html - visualisation des données montrant comment le GIL verrouille les threads

http://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/ - simple à comprendre l'histoire du problème GIL

https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/ - des détails sur la manière de contourner les limitations du GIL

Examples

Multiprocessing.Pool

La réponse simple à la question de savoir comment utiliser les threads dans Python est la suivante: "Ne le faites pas. Utilisez plutôt des processus." Le module de multitraitement vous permet de créer des processus avec une syntaxe similaire à la création de threads, mais je préfère utiliser leur objet Pool pratique.

En utilisant le code que David Beazley a d'abord utilisé pour montrer les dangers des threads sur le GIL , nous allons le réécrire à l'aide du multitraitement .

Le code de David Beazley qui a montré des problèmes de thread GIL

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = Thread(target=countdown,args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Re-écrit en utilisant multiprocessing.Pool:

```
import multiprocessing
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

start = time.time()
with multiprocessing.Pool as pool:
    pool.map(countdown, [COUNT/2, COUNT/2])

    pool.close()
    pool.join()

end = time.time()
print(end-start)
```

Au lieu de créer des threads, cela crée de nouveaux processus. Comme chaque processus est son propre interpréteur, il n'y a pas de collision de GIL. multiprocessing.Pool ouvrira autant de processus qu'il y a de cœurs sur la machine, bien que dans l'exemple ci-dessus, il n'en faudrait que deux. Dans un scénario réel, vous souhaitez concevoir votre liste de manière à avoir au moins autant de longueur que des processeurs sur votre machine. Le pool exécutera la fonction que vous lui indiquez pour chaque argument, jusqu'au nombre de processus qu'il crée. Lorsque la fonction se termine, toutes les fonctions restantes dans la liste seront exécutées sur ce processus.

J'ai trouvé que, même en utilisant l'instruction `with` , si vous ne fermez pas et ne rejoignez pas le pool, les processus continuent d'exister. Pour nettoyer les ressources, je ferme et joignais toujours mes pools.

Cython nogil:

Cython est un interpréteur Python alternatif. Il utilise le GIL, mais vous permet de le désactiver. Voir leur documentation

À titre d'exemple, en utilisant le code que David Beazley a utilisé pour montrer les dangers des threads contre le GIL , nous allons le réécrire en utilisant nogil:

Le code de David Beazley qui a montré des problèmes de thread GIL

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = Thread(target=countdown,args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Réécrit en utilisant nogil (SEULEMENT FONCTIONNE À CYTHON):

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
```

```
        n -= 1

COUNT = 10000000

with nogil:
    t1 = Thread(target=countdown,args=(COUNT/2,))
    t2 = Thread(target=countdown,args=(COUNT/2,))
    start = time.time()
    t1.start();t2.start()
    t1.join();t2.join()

end = time.time()
print end-start
```

C'est si simple, tant que vous utilisez cython. Notez que la documentation indique que vous devez vous assurer de ne modifier aucun objet python:

> Le code dans le corps de l'instruction ne doit en aucun cas manipuler les objets Python, et ne doit rien appeler qui manipule les objets Python sans avoir préalablement racheté le GIL. Cython ne vérifie pas actuellement cela.

Lire Travailler autour du verrou d'interprète global (GIL) en ligne: https://riptutorial.com/fr/python/topic/4061/travailler-autour-du-verrou-d-interprete-global--gil-

Chapitre 191: Travailler avec des archives ZIP

Syntaxe

- importer le fichier zip
- fichier zip de *classe* . **ZipFile** (*fichier, mode = 'r', compression = ZIP_STORED, allowZip64 = True*)

Remarques

Si vous essayez d'ouvrir un fichier qui n'est pas un fichier ZIP, l'exception `zipfile.BadZipFile` est `zipfile.BadZipFile` .

Dans Python 2.7, ceci était orthographié `zipfile.BadZipfile` , et cet ancien nom est conservé à côté du nouveau dans Python 3.2+

Examples

Ouverture de fichiers Zip

Pour commencer, importez le module `zipfile` et définissez le nom du fichier.

```
import zipfile
filename = 'zipfile.zip'
```

Travailler avec des archives zip est très similaire à travailler avec des fichiers , vous créez l'objet en ouvrant le fichier zip, ce qui vous permet de travailler dessus avant de refermer le fichier.

```
zip = zipfile.ZipFile(filename)
print(zip)
# <zipfile.ZipFile object at 0x0000000002E51A90>
zip.close()
```

Dans Python 2.7 et dans les versions Python 3 supérieures à 3.2, nous pouvons utiliser le gestionnaire de contexte `with` . Nous ouvrons le fichier en mode "read", puis imprimons une liste de noms de fichiers:

```
with zipfile.ZipFile(filename, 'r') as z:
    print(zip)
    # <zipfile.ZipFile object at 0x0000000002E51A90>
```

Examen du contenu du fichier zip

Il y a plusieurs manières d'inspecter le contenu d'un fichier zip. Vous pouvez utiliser le `printdir` pour obtenir une variété d'informations envoyées à `stdout`

```
with zipfile.ZipFile(filename) as zip:
    zip.printdir()

    # Out:
    # File Name                                             Modified             Size
    # pyexpat.pyd                                    2016-06-25 22:13:34       157336
    # python.exe                                     2016-06-25 22:13:34        39576
    # python3.dll                                    2016-06-25 22:13:34        51864
    # python35.dll                                   2016-06-25 22:13:34      3127960
    # etc.
```

Nous pouvons également obtenir une liste de noms de fichiers avec la méthode `namelist` . Ici, nous imprimons simplement la liste:

```
with zipfile.ZipFile(filename) as zip:
    print(zip.namelist())

# Out: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... etc. ...]
```

Au lieu de la `namelist` de `namelist` , nous pouvons appeler la méthode `infolist` , qui renvoie une liste d'objets `ZipInfo` , qui contiennent des informations supplémentaires sur chaque fichier, par exemple un horodatage et une taille de fichier:

```
with zipfile.ZipFile(filename) as zip:
    info = zip.infolist()
    print(zip[0].filename)
    print(zip[0].date_time)
    print(info[0].file_size)

# Out: pyexpat.pyd
# Out: (2016, 6, 25, 22, 13, 34)
# Out: 157336
```

Extraire le contenu d'un fichier zip dans un répertoire

Extraire tout le contenu d'un fichier zip

```
import zipfile
with zipfile.ZipFile('zipfile.zip','r') as zfile:
    zfile.extractall('path')
```

Si vous souhaitez extraire des fichiers uniques, utilisez la méthode d'extraction, il prend la liste de noms et le chemin comme paramètre d'entrée

```
import zipfile
f=open('zipfile.zip','rb')
zfile=zipfile.ZipFile(f)
for cont in zfile.namelist():
    zfile.extract(cont,path)
```

Créer de nouvelles archives

Pour créer une nouvelle archive, ouvrez le fichier zip avec le mode écriture.

```
import zipfile
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

Pour ajouter des fichiers à cette archive, utilisez la méthode write ().

```
new_arch.write('filename.txt','filename_in_archive.txt') #first parameter is filename and
second parameter is filename in archive by default filename will taken if not provided
new_arch.close()
```

Si vous voulez écrire une chaîne d'octets dans l'archive, vous pouvez utiliser la méthode writestr ().

```
str_bytes="string buffer"
new_arch.writestr('filename_string_in_archive.txt',str_bytes)
new_arch.close()
```

Lire Travailler avec des archives ZIP en ligne: https://riptutorial.com/fr/python/topic/3728/travailler-avec-des-archives-zip

Chapitre 192: Tri, minimum et maximum

Examples

Obtenir le minimum ou le maximum de plusieurs valeurs

```
min(7,2,1,5)
# Output: 1

max(7,2,1,5)
# Output: 7
```

Utiliser l'argument clé

Trouver le minimum / maximum d'une séquence de séquences est possible:

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Output: (0, 10)
```

mais si vous voulez trier par un élément spécifique dans chaque séquence, utilisez l'argument `key` :

```
min(list_of_tuples, key=lambda x: x[0])         # Sorting by first element
# Output: (0, 10)

min(list_of_tuples, key=lambda x: x[1])         # Sorting by second element
# Output: (2, 8)

sorted(list_of_tuples, key=lambda x: x[0])      # Sorting by first element (increasing)
# Output: [(0, 10), (1, 15), (2, 8)]

sorted(list_of_tuples, key=lambda x: x[1])      # Sorting by first element
# Output: [(2, 8), (0, 10), (1, 15)]

import operator
# The operator module contains efficient alternatives to the lambda function
max(list_of_tuples, key=operator.itemgetter(0)) # Sorting by first element
# Output: (2, 8)

max(list_of_tuples, key=operator.itemgetter(1)) # Sorting by second element
# Output: (1, 15)

sorted(list_of_tuples, key=operator.itemgetter(0), reverse=True) # Reversed (decreasing)
# Output: [(2, 8), (1, 15), (0, 10)]

sorted(list_of_tuples, key=operator.itemgetter(1), reverse=True) # Reversed(decreasing)
# Output: [(1, 15), (0, 10), (2, 8)]
```

Argument par défaut à max, min

Vous ne pouvez pas passer une séquence vide dans `max` ou `min` :

```
min([])
```

ValueError: min () arg est une séquence vide

Cependant, avec Python 3, vous pouvez transmettre l'argument `default` avec une valeur qui sera renvoyée si la séquence est vide, au lieu de générer une exception:

```
max([], default=42)
# Output: 42
max([], default=0)
# Output: 0
```

Cas particulier: dictionnaires

Le minimum ou le maximum ou l'utilisation de `sorted` dépend des itérations sur l'objet. Dans le cas de `dict` , l'itération est uniquement sur les clés:

```
adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Output: 'a'
max(adict)
# Output: 'c'
sorted(adict)
# Output: ['a', 'b', 'c']
```

Pour conserver la structure du dictionnaire, vous devez parcourir le `.items()` :

```
min(adict.items())
# Output: ('a', 3)
max(adict.items())
# Output: ('c', 1)
sorted(adict.items())
# Output: [('a', 3), ('b', 5), ('c', 1)]
```

Pour `sorted` , vous pouvez créer un `OrderedDict` pour conserver le tri tout en ayant une structure de type `dict` :

```
from collections import OrderedDict
OrderedDict(sorted(adict.items()))
# Output: OrderedDict([('a', 3), ('b', 5), ('c', 1)])
res = OrderedDict(sorted(adict.items()))
res['a']
# Output: 3
```

Par valeur

Là encore, cela est possible en utilisant l'argument `key` :

```
min(adict.items(), key=lambda x: x[1])
# Output: ('c', 1)
max(adict.items(), key=operator.itemgetter(1))
# Output: ('b', 5)
sorted(adict.items(), key=operator.itemgetter(1), reverse=True)
# Output: [('b', 5), ('a', 3), ('c', 1)]
```

Obtenir une séquence triée

En utilisant **une** séquence:

```
sorted((7, 2, 1, 5))                 # tuple
# Output: [1, 2, 5, 7]

sorted(['c', 'A', 'b'])              # list
# Output: ['A', 'b', 'c']

sorted({11, 8, 1})                   # set
# Output: [1, 8, 11]

sorted({'11': 5, '3': 2, '10': 15})  # dict
# Output: ['10', '11', '3']          # only iterates over the keys

sorted('bdca')                       # string
# Output: ['a','b','c','d']
```

Le résultat est toujours une nouvelle `list` ; les données d'origine restent inchangées.

Minimum et Maximum d'une séquence

Obtenir le minimum d'une séquence (itérable) équivaut à accéder au premier élément d'une séquence `sorted` :

```
min([2, 7, 5])
# Output: 2
sorted([2, 7, 5])[0]
# Output: 2
```

Le maximum est un peu plus compliqué, car `sorted` garde l'ordre et `max` renvoie la première valeur rencontrée. S'il n'y a pas de doublons, le maximum est le même que le dernier élément du retour trié:

```
max([2, 7, 5])
# Output: 7
sorted([2, 7, 5])[-1]
# Output: 7
```

Mais pas si plusieurs éléments sont évalués comme ayant la valeur maximale:

```
class MyClass(object):
    def __init__(self, value, name):
        self.value = value
        self.name = name
```

```
    def __lt__(self, other):
        return self.value < other.value

    def __repr__(self):
        return str(self.name)

sorted([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: [second, first, third]
max([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: first
```

Tout élément contenant des éléments pouvant supporter `<` ou `>` opérations est autorisé.

Rendre les classes personnalisées ordonnables

`min` , `max` et `sorted` doivent tous les objets être ordonnables. Pour être correctement classable, la classe doit définir toutes les 6 méthodes `__lt__` , `__gt__` , `__ge__` , `__le__` , `__ne__` et `__eq__` :

```
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __le__(self, other):
        print('{!r} - Test less than or equal to {!r}'.format(self, other))
        return self.value <= other.value

    def __gt__(self, other):
        print('{!r} - Test greater than {!r}'.format(self, other))
        return self.value > other.value

    def __ge__(self, other):
        print('{!r} - Test greater than or equal to {!r}'.format(self, other))
        return self.value >= other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value
```

Bien que l'implémentation de toutes ces méthodes semble inutile, en omettre certaines rendra votre code sujet aux bogues .

Exemples:

```
alist = [IntegerContainer(5), IntegerContainer(3),
```

```
         IntegerContainer(10), IntegerContainer(7)
        ]

res = max(alist)
# Out: IntegerContainer(3) - Test greater than IntegerContainer(5)
#      IntegerContainer(10) - Test greater than IntegerContainer(5)
#      IntegerContainer(7) - Test greater than IntegerContainer(10)
print(res)
# Out: IntegerContainer(10)

res = min(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
#      IntegerContainer(7) - Test less than IntegerContainer(3)
print(res)
# Out: IntegerContainer(3)

res = sorted(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
#      IntegerContainer(10) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(10)
print(res)
# Out: [IntegerContainer(3), IntegerContainer(5), IntegerContainer(7), IntegerContainer(10)]
```

`sorted` avec `reverse=True` utilise également `__lt__` :

```
res = sorted(alist, reverse=True)
# Out: IntegerContainer(10) - Test less than IntegerContainer(7)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(3)
print(res)
# Out: [IntegerContainer(10), IntegerContainer(7), IntegerContainer(5), IntegerContainer(3)]
```

Mais `sorted` peut utiliser `__gt__` place si la valeur par défaut n'est pas implémentée:

```
del IntegerContainer.__lt__   # The IntegerContainer no longer implements "less than"

res = min(alist)
# Out: IntegerContainer(5) - Test greater than IntegerContainer(3)
#      IntegerContainer(3) - Test greater than IntegerContainer(10)
#      IntegerContainer(3) - Test greater than IntegerContainer(7)
print(res)
# Out: IntegerContainer(3)
```

Les méthodes de tri `TypeError` une `TypeError` si ni `__lt__` ni `__gt__` sont implémentées:

```
del IntegerContainer.__gt__   # The IntegerContainer no longer implements "greater then"

res = min(alist)
```

TypeError: types non ordonnables: IntegerContainer () <IntegerContainer ()

`functools.total_ordering` décorateur peut être utilisé pour simplifier l'écriture de ces riches méthodes de comparaison. Si vous décorez votre classe avec `total_ordering` , vous devez implémenter `__eq__` , `__ne__` et un seul des `__lt__` , `__le__` , `__ge__` ou `__gt__` , et le décorateur remplira le reste:

```
import functools

@functools.total_ordering
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value


IntegerContainer(5) > IntegerContainer(6)
# Output: IntegerContainer(5) - Test less than IntegerContainer(6)
# Returns: False

IntegerContainer(6) > IntegerContainer(5)
# Output: IntegerContainer(6) - Test less than IntegerContainer(5)
# Output: IntegerContainer(6) - Test equal to IntegerContainer(5)
# Returns True
```

Notez que le `>` (*supérieur à*) finit maintenant par appeler la méthode *less than* et, dans certains cas, même la méthode `__eq__` . Cela signifie également que si la vitesse est très importante, vous devez implémenter vous-même chaque méthode de comparaison enrichie.

Extraire N plus grand ou N plus petit élément d'une

Pour trouver un nombre (plus d'un) de la plus grande ou de la plus petite valeur d'une itération, vous pouvez utiliser le `nlargest` et le `nsmallest` du module `heapq` :

```
import heapq

# get 5 largest items from the range

heapq.nlargest(5, range(10))
# Output: [9, 8, 7, 6, 5]

heapq.nsmallest(5, range(10))
# Output: [0, 1, 2, 3, 4]
```

Ceci est beaucoup plus efficace que de trier la totalité des itérables puis de les couper de la fin ou du début. En interne, ces fonctions utilisent la structure de données de la file d'attente du segment de mémoire binaire , ce qui est très efficace pour ce cas d'utilisation.

Comme `min` , `max` et `sorted` , ces fonctions acceptent l'option `key` argument de mot - clé, qui doit être une fonction qui, étant donné un élément, renvoie sa clé de tri.

Voici un programme qui extrait 1000 lignes les plus longues d'un fichier:

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

Ici, nous ouvrons le fichier et transmettons le `nlargest` fichier `f` à la plus grande `nlargest` . Itérer le fichier donne à chaque ligne du fichier une chaîne distincte; `nlargest` passe ensuite chaque élément (ou ligne) est passé à la fonction `len` pour déterminer sa clé de tri. `len` , étant donné une chaîne, renvoie la longueur de la ligne en caractères.

Cela ne nécessite que le stockage pour une liste de 1000 lignes les plus grandes à ce jour, qui peut être comparée à

```
longest_lines = sorted(f, key=len)[1000:]
```

qui devra contenir *tout le fichier en mémoire* .

Lire Tri, minimum et maximum en ligne: https://riptutorial.com/fr/python/topic/252/tri--minimum-et-maximum

Chapitre 193: Tuple

Introduction

Un tuple est une liste de valeurs immuable. Les tuples sont l'un des types de collection les plus simples et les plus communs de Python et peuvent être créés avec l'opérateur virgule (`value = 1, 2, 3`).

Syntaxe

- (1, a, "hello") # a doit être une variable
- () # un tuple vide
- (1) # un tuple à 1 élément. `(1)` n'est pas un tuple.
- 1, 2, 3 # le tuple à 3 éléments (1, 2, 3)

Remarques

Les parenthèses ne sont nécessaires que pour les tuples vides ou lorsqu'ils sont utilisés dans un appel de fonction.

Un tuple est une séquence de valeurs. Les valeurs peuvent être de n'importe quel type, et elles sont indexées par des nombres entiers, de sorte que les n-uplets ressemblent beaucoup aux listes. La différence importante est que les tuples sont immuables et sont lavables, ils peuvent donc être utilisés dans des ensembles et des cartes.

Examples

Tuples d'indexation

```
x = (1, 2, 3)
x[0]  # 1
x[1]  # 2
x[2]  # 3
x[3]  # IndexError: tuple index out of range
```

L'indexation avec des nombres négatifs commencera à partir du dernier élément comme -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Indexer une gamme d'éléments

```
print(x[:-1])   # (1, 2)
print(x[-1:])   # (3,)
print(x[1:3])   # (2, 3)
```

Les tuples sont immuables

L'une des principales différences entre les `list` s et les `tuple` dans Python est que les tuples sont immuables, c'est-à-dire que l'on ne peut pas ajouter ou modifier des éléments une fois le tuple initialisé. Par exemple:

```
>>> t = (1, 4, 9)
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

De même, les tuples n'ont pas de méthodes `.append` et `.extend` comme le fait la `list` . Utiliser `+=` est possible, mais cela change la liaison de la variable, et non le tuple lui-même:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Soyez prudent lorsque vous placez des objets mutables, tels que des `lists` , à l'intérieur de tuples. Cela peut conduire à des résultats très déroutants lors de leur modification. Par exemple:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Est-ce que les ***deux*** déclencheront une erreur et changeront le contenu de la liste dans le tuple:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

Vous pouvez utiliser l'opérateur `+=` pour "ajouter" à un tuple - cela fonctionne en créant un nouveau tuple avec le nouvel élément "ajouté" et l'assignez à sa variable actuelle; le vieux tuple n'est pas changé, mais remplacé!

Cela évite la conversion vers et depuis une liste, mais c'est lent et c'est une mauvaise pratique, surtout si vous allez ajouter plusieurs fois.

Le tuple est élémentaire et lavable

```
hash( (1, 2) )  # ok
```

```
hash( ([], {"hello"})  # not ok, since lists and sets are not hashabe
```

Ainsi, un tuple ne peut être placé dans un `set` ou comme clé dans un `dict` que si chacun de ses éléments le peut.

```
{ (1, 2) } #  ok
{ ([], {"hello"}) ) # not ok
```

Tuple

Syntactiquement, un tuple est une liste de valeurs séparées par des virgules:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Bien que cela ne soit pas nécessaire, il est courant de joindre des tuples entre parenthèses:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Créez un nuplet vide avec des parenthèses:

```
t0 = ()
type(t0)            # <type 'tuple'>
```

Pour créer un tuple avec un seul élément, vous devez inclure une virgule finale:

```
t1 = 'a',
type(t1)              # <type 'tuple'>
```

Notez qu'une valeur unique entre parenthèses n'est pas un tuple:

```
t2 = ('a')
type(t2)              # <type 'str'>
```

Pour créer un tuple singleton, il est nécessaire d'avoir une virgule de fin.

```
t2 = ('a',)
type(t2)              # <type 'tuple'>
```

Notez que pour les tuples singleton, il est recommandé d'utiliser les parenthèses (voir PEP8 sur les virgules finales). Aussi, pas d'espace blanc après la virgule de fin (voir PEP8 sur les espaces blancs)

```
t2 = ('a',)           # PEP8-compliant
t2 = 'a',             # this notation is not recommended by PEP8
t2 = ('a', )          # this notation is not recommended by PEP8
```

Une autre façon de créer un tuple est le `tuple` fonction intégré.

```
t = tuple('lupins')
print(t)              # ('l', 'u', 'p', 'i', 'n', 's')
t = tuple(range(3))
print(t)              # (0, 1, 2)
```

Ces exemples sont basés sur le matériel du livre **Think Python** d'Allen B. Downey .

Emballage et déballage des tuples

Les tuples en Python sont des valeurs séparées par des virgules. La mise entre parenthèses pour la saisie de n-uplets est facultative, donc les deux affectations

```
a = 1, 2, 3   # a is the tuple (1, 2, 3)
```

et

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

sont équivalents. L'affectation `a = 1, 2, 3` est également appelée " *emballage"* car elle regroupe les valeurs dans un tuple.

Notez qu'un tuple à une valeur est également un tuple. Pour dire à Python qu'une variable est un tuple et non une valeur unique, vous pouvez utiliser une virgule

```
a = 1  # a is the value 1
a = 1, # a is the tuple (1,)
```

Une virgule est également nécessaire si vous utilisez des parenthèses

```
a = (1,) # a is the tuple (1,)
a = (1)  # a is the value 1 and not a tuple
```

Pour décompresser les valeurs d'un tuple et faire plusieurs affectations, utilisez

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

Le symbole _ peut être utilisé comme un nom de variable jetable si l'on a seulement besoin de quelques éléments d'un tuple, agissant comme un espace réservé:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Tuples à élément unique:

```
x, = 1,  # x is the value 1
x  = 1,  # x is the tuple (1,)
```

Dans Python 3, une variable cible avec un préfixe `*` peut être utilisée comme variable *fourre-tout* (voir Déballage d'itérables):

Python 3.x 3.0

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

Éléments d'inversion

Inverser les éléments dans un tuple

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Ou en utilisant inversé (inversé donne un itérable qui est converti en un tuple):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Fonctions Tuple intégrées

Les tuples supportent les fonctions intégrées suivantes

Comparaison

Si les éléments sont du même type, python effectue la comparaison et renvoie le résultat. Si les éléments sont de types différents, il vérifie s'ils sont des nombres.

- Si les nombres, effectuez une comparaison.
- Si l'un des éléments est un nombre, l'autre élément est renvoyé.
- Sinon, les types sont triés par ordre alphabétique.

Si nous avons atteint la fin de l'une des listes, la liste la plus longue est "plus grande". Si les deux listes sont identiques, il renvoie 0.

```
tuple1 = ('a', 'b', 'c', 'd', 'e')
tuple2 = ('1','2','3')
tuple3 = ('a', 'b', 'c', 'd', 'e')
```

```
cmp(tuple1, tuple2)
Out: 1

cmp(tuple2, tuple1)
Out: -1

cmp(tuple1, tuple3)
Out: 0
```

Longueur de tuple

La fonction `len` renvoie la longueur totale du tuple

```
len(tuple1)
Out: 5
```

Max d'un tuple

La fonction `max` renvoie un élément du tuple avec la valeur maximale

```
max(tuple1)
Out: 'e'

max(tuple2)
Out: '3'
```

Min d'un tuple

La fonction min renvoie l'élément du tuple avec la valeur min

```
min(tuple1)
Out: 'a'

min(tuple2)
Out: '1'
```

Convertir une liste en tuple

Le `tuple` fonction intégré convertit une liste en un tuple.

```
list = [1,2,3,4,5]
tuple(list)
Out: (1, 2, 3, 4, 5)
```

Concaténation tuple

Utilisez `+` pour concaténer deux tuples

```
tuple1 + tuple2
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

Lire Tuple en ligne: https://riptutorial.com/fr/python/topic/927/tuple

Chapitre 194: Type conseils

Syntaxe

- typing.Callable [[int, str], None] -> def func (a: int, b: str) -> Aucun
- typing.Mapping [str, int] -> {"a": 1, "b": 2, "c": 3}
- typing.List [int] -> [1, 2, 3]
- typing.Set [int] -> {1, 2, 3}
- typing.Optional [int] -> Aucun ou int
- typing.Sequence [int] -> [1, 2, 3] ou (1, 2, 3)
- taper.Any -> N'importe quel type
- taper.Union [int, str] -> 1 ou "1"
- T = typing.TypeVar ('T') -> Type générique

Remarques

L'indication de type, spécifiée dans `PEP 484` , est une solution formalisée permettant d'indiquer statiquement le type d'une valeur pour le code Python. En apparaissant à côté du module de `typing` , les `typing` type offrent aux utilisateurs Python la possibilité d'annoter leur code, aidant ainsi les vérificateurs de type tout en documentant indirectement leur code avec plus d'informations.

Examples

Types génériques

Le `typing.TypeVar` est une fabrique de type générique. Son objectif principal est de servir de paramètre / espace réservé pour les annotations génériques de fonction / classe / méthode:

```
import typing

T = typing.TypeVar("T")

def get_first_element(l: typing.Sequence[T]) -> T:
    """Gets the first element of a sequence."""
    return l[0]
```

Ajouter des types à une fonction

Prenons un exemple de fonction qui reçoit deux arguments et renvoie une valeur indiquant leur somme:

```
def two_sum(a, b):
    return a + b
```

En regardant ce code, on ne peut pas et sans doute indiquer le type des arguments pour la fonction `two_sum` . Il fonctionne à la fois avec les valeurs `int` :

```
print(two_sum(2, 1))  # result: 3
```

et avec des cordes:

```
print(two_sum("a", "b"))  # result: "ab"
```

et avec d'autres valeurs, telles que les `list` s, `tuple` s et cetera.

En raison de la nature dynamique des types python, où beaucoup sont applicables à une opération donnée, tout vérificateur de type ne pourrait pas raisonnablement affirmer si un appel pour cette fonction devrait être autorisé ou non.

Pour aider notre vérificateur de types, nous pouvons maintenant lui fournir des indications de type dans la définition de fonction, indiquant le type que nous autorisons.

Pour indiquer que nous voulons seulement autoriser les types `int` nous pouvons changer notre définition de fonction pour qu'elle ressemble à:

```
def two_sum(a: int, b: int):
    return a + b
```

Les annotations suivent le nom de l' argument et sont séparés par un `:` caractère.

De même, pour indiquer que seuls les types `str` sont autorisés, nous modifierons notre fonction pour la spécifier:

```
def two_sum(a: str, b: str):
    return a + b
```

En plus de spécifier le type des arguments, on pourrait également indiquer la valeur de retour d'un appel de fonction. Cela se fait en ajoutant le caractère `->` suivi du type après la parenthèse fermante dans la liste des arguments, *mais* avant le `:` à la fin de la déclaration de la fonction:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

Maintenant, nous avons indiqué que la valeur de retour lors de l'appel à `two_sum` devrait être de type `int` . De même, nous pouvons définir des valeurs appropriées pour `str` , `float` , `list` , `set` et autres.

Bien que les indicateurs de type soient principalement utilisés par les vérificateurs de type et les IDE, il est parfois nécessaire de les récupérer. Cela peut être fait en utilisant l' `__annotations__` spécial `__annotations__` :

```
two_sum.__annotations__
```

```
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Membres de la classe et méthodes

```
class A:
    x = None  # type: float
    def __init__(self, x: float) -> None:
        """
        self should not be annotated
        init should be annotated to return None
        """
        self.x = x

    @classmethod
    def from_int(cls, x: int) -> 'A':
        """
        cls should not be annotated
        Use forward reference to refer to current class with string literal 'A'
        """
        return cls(float(x))
```

La référence en aval de la classe en cours est nécessaire car les annotations sont évaluées lorsque la fonction est définie. Les références directes peuvent également être utilisées lorsque vous faites référence à une classe qui provoquerait une importation circulaire si elle était importée.

Variables et attributs

Les variables sont annotées en utilisant les commentaires:

```
x = 3  # type: int
x = negate(x)
x = 'a type-checker might catch this error'
```

Python 3.x 3.6

À partir de Python 3.6, il existe également une nouvelle syntaxe pour les annotations de variables . Le code ci-dessus pourrait utiliser le formulaire

```
x: int = 3
```

Contrairement aux commentaires, il est également possible d'ajouter un indice de type à une variable qui n'a pas été déclarée précédemment, sans lui donner de valeur:

```
y: int
```

En outre, si elles sont utilisées au niveau du module ou de la classe, les indications de type peuvent être récupérées à l'aide de `typing.get_type_hints(class_or_module)` :

```
class Foo:
    x: int
    y: str = 'abc'
```

```
print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

Vous pouvez également y accéder en utilisant la variable ou l'attribut spécial `__annotations__` :

```
x: int
print(__annotations__)
# {'x': <class 'int'>}

class C:
    s: str
print(C.__annotations__)
# {'s': <class 'str'>}
```

NomméTuple

La création d'un élément nommé avec des indications de type se fait à l'aide de la fonction NamedTuple du module de `typing` :

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

Notez que le nom du type résultant est le premier argument de la fonction, mais qu'il doit être affecté à une variable du même nom pour faciliter le travail des vérificateurs de type.

Indiquer des astuces pour les arguments de mots clés

```
def hello_world(greeting: str = 'Hello'):
    print(greeting + ' world!')
```

Notez les espaces autour du signe égal, par opposition à la façon dont les arguments sont généralement stylés.

Lire Type conseils en ligne: https://riptutorial.com/fr/python/topic/1766/type-conseils

Chapitre 195: Types de données immuables (int, float, str, tuple et frozensets)

Examples

Les caractères individuels des chaînes ne sont pas assignables

```
foo = "bar"
foo[0] = "c" # Error
```

La valeur de la variable immuable ne peut pas être modifiée une fois créée.

Les membres individuels de Tuple ne sont pas assignables

```
foo = ("bar", 1, "Hello!",)
foo[1] = 2 # ERROR!!
```

La deuxième ligne renverrait une erreur car les membres du tuple une fois créés ne sont pas assignables. A cause de l'immuabilité du tuple.

Les Frozenset sont immuables et non assignables

```
foo = frozenset(["bar", 1, "Hello!"])
foo[2] = 7 # ERROR
foo.add(3) # ERROR
```

La deuxième ligne renverrait une erreur car les membres de frozenset une fois créés ne sont pas assignables. La troisième ligne renverrait une erreur car les frozensets ne supportent pas les fonctions pouvant manipuler les membres.

Lire Types de données immuables (int, float, str, tuple et frozensets) en ligne: https://riptutorial.com/fr/python/topic/4806/types-de-donnees-immuables--int--float--str--tuple-et-frozensets-

Chapitre 196: Types de données Python

Introduction

Les types de données ne sont que des variables que vous avez utilisées pour réserver de l'espace en mémoire. Les variables Python n'ont pas besoin d'une déclaration explicite pour réserver de l'espace mémoire. La déclaration se produit automatiquement lorsque vous attribuez une valeur à une variable.

Examples

Type de données de nombres

Les nombres ont quatre types en Python. Int, float, complex et long.

```
int_num = 10    #int value
float_num = 10.2    #float value
complex_num = 3.14j    #complex value
long_num = 1234567L    #long value
```

Type de données de chaîne

Les chaînes sont identifiées comme un ensemble de caractères contigus représenté par les guillemets. Python permet des paires de guillemets simples ou doubles. Les chaînes sont des types de données de séquence immuables, c'est-à-dire que chaque fois que l'on modifie une chaîne, un objet chaîne complètement nouveau est créé.

```
a_str = 'Hello World'
print(a_str)    #output will be whole string. Hello World
print(a_str[0])    #output will be first character. H
print(a_str[0:5])    #output will be first five characters. Hello
```

Type de données de liste

Une liste contient des éléments séparés par des virgules et entre crochets []. Les listes sont presque similaires aux tableaux de C. Une différence est que tous les éléments appartenant à une liste peuvent être de type de données différent.

```
list = [123,'abcd',10.2,'d']    #can be a array of any data type or single data type.
list1 = ['hello','world']
print(list)    #will ouput whole list. [123,'abcd',10.2,'d']
print(list[0:2])    #will output first two element of list. [123,'abcd']
print(list1 * 2)    #will gave list1 two times. ['hello','world','hello','world']
print(list + list1)    #will gave concatenation of both the lists.
[123,'abcd',10.2,'d','hello','world']
```

Type de données Tuple

Les listes sont placées entre parenthèses [] et leurs éléments et leur taille peuvent être modifiés, tandis que les tuples sont entre parenthèses () et ne peuvent pas être mis à jour. Les tuples sont immuables.

```
tuple = (123,'hello')
tuple1 = ('world')
print(tuple)    #will output whole tuple. (123,'hello')
print(tuple[0])    #will output first value. (123)
print(tuple + tuple1)    #will output (123,'hello','world')
tuple[1]='update'    #this will give you error.
```

Type de données du dictionnaire

Le dictionnaire se compose de paires clé-valeur. Il est encadré par des accolades {} et les valeurs peuvent être attribuées et utilisées entre crochets [].

```
dic={'name':'red','age':10}
print(dic)    #will output all the key-value pairs. {'name':'red','age':10}
print(dic['name'])    #will output only value with 'name' key. 'red'
print(dic.values())    #will output list of values in dic. ['red',10]
print(dic.keys())    #will output list of keys. ['name','age']
```

Définir les types de données

Les ensembles sont des collections non ordonnées d'objets uniques, il existe deux types d'ensembles:

1. Sets - Ils sont mutables et de nouveaux éléments peuvent être ajoutés une fois les ensembles définis

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)            # duplicates will be removed
> {'orange', 'banana', 'pear', 'apple'}
a = set('abracadabra')
print(a)                 # unique letters in a
> {'a', 'r', 'b', 'c', 'd'}
a.add('z')
print(a)
> {'a', 'c', 'r', 'b', 'z', 'd'}
```

2. Ensembles gelés - Ils sont immuables et les nouveaux éléments ne peuvent pas être ajoutés après leur définition.

```
b = frozenset('asdfagsa')
print(b)
> frozenset({'f', 'g', 'd', 'a', 's'})
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
print(cities)
> frozenset({'Frankfurt', 'Basel', 'Freiburg'})
```

Lire Types de données Python en ligne: https://riptutorial.com/fr/python/topic/9366/types-de-donnees-python

Chapitre 197: Unicode

Examples

Encodage et décodage

Toujours *encoder* d'Unicode en octets. Dans cette direction, **vous choisissez le codage** .

```
>>> u'🐍'.encode('utf-8')
'\xf0\x9f\x90\x8d'
```

L'autre méthode consiste à *décoder les* octets en unicode. Dans cette direction, **vous devez savoir quel est le codage** .

```
>>> b'\xf0\x9f\x90\x8d'.decode('utf-8')
u'\U0001f40d'
```

Lire Unicode en ligne: https://riptutorial.com/fr/python/topic/5618/unicode

Chapitre 198: Unicode et octets

Syntaxe

- str.encode (encodage, erreurs = 'strict')
- bytes.decode (encodage, erreurs = 'strict')
- ouvrir (nom de fichier, mode, codage = Aucun)

Paramètres

Paramètre	Détails
codage	Le codage à utiliser, par exemple `'ascii'` , `'utf8'` , etc ...
les erreurs	Le mode des erreurs, par exemple `'replace'` pour remplacer les mauvais caractères par des points d'interrogation, `'ignore'` pour ignorer les mauvais caractères, etc ...

Examples

Les bases

Dans Python 3, `str` est le type des chaînes activées par Unicode, tandis que les `bytes` sont le type des séquences d'octets bruts.

```
type("f") == type(u"f")   # True, <class 'str'>
type(b"f")                # <class 'bytes'>
```

Dans Python 2, une chaîne occasionnelle était une séquence d'octets bruts par défaut et la chaîne unicode correspondait à chaque chaîne avec le préfixe "u".

```
type("f") == type(b"f")   # True, <type 'str'>
type(u"f")                # <type 'unicode'>
```

Unicode en octets

Les chaînes Unicode peuvent être converties en octets avec `.encode(encoding)` .

Python 3

```
>>> "£13.55".encode('utf8')
```

```
b'\xc2\xa313.55'
>>> "£13.55".encode('utf16')
b'\xff\xfe\xa3\x001\x003\x00.\x005\x005\x00'
```

Python 2

dans py2, l'encodage par défaut de la console est `sys.getdefaultencoding() == 'ascii'` et non `utf-8` comme dans py3, par conséquent, l'impression comme dans l'exemple précédent n'est pas directement possible.

```
>>> print type(u"£13.55".encode('utf8'))
<type 'str'>
>>> print u"£13.55".encode('utf8')
SyntaxError: Non-ASCII character '\xc2' in...

# with encoding set inside a file

# -*- coding: utf-8 -*-
>>> print u"£13.55".encode('utf8')
┬ú13.55
```

Si l'encodage ne peut pas gérer la chaîne, un `UnicodeEncodeError` est généré:

```
>>> "£13.55".encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xa3' in position 0: ordinal not in
range(128)
```

Octets à unicode

Les octets peuvent être convertis en chaînes Unicode avec `.decode(encoding)` .

Une séquence d'octets ne peut être convertie en une chaîne Unicode que via le codage approprié!

```
>>> b'\xc2\xa313.55'.decode('utf8')
'£13.55'
```

Si le codage ne peut pas gérer la chaîne, une `UnicodeDecodeError` est `UnicodeDecodeError` :

```
>>> b'\xc2\xa313.55'.decode('utf16')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/csaftoiu/csaftoiu-github/yahoo-groups-
backup/.virtualenv/bin/../lib/python3.5/encodings/utf_16.py", line 16, in decode
    return codecs.utf_16_decode(input, errors, True)
UnicodeDecodeError: 'utf-16-le' codec can't decode byte 0x35 in position 6: truncated data
```

Gestion des erreurs d'encodage / décodage

`.encode` et `.decode` ont tous deux des modes d'erreur.

La valeur par défaut est `'strict'` , ce qui génère des exceptions en cas d'erreur. Les autres modes sont plus tolérants.

Codage

```
>>> "£13.55".encode('ascii', errors='replace')
b'?13.55'
>>> "£13.55".encode('ascii', errors='ignore')
b'13.55'
>>> "£13.55".encode('ascii', errors='namereplace')
b'\\N{POUND SIGN}13.55'
>>> "£13.55".encode('ascii', errors='xmlcharrefreplace')
b'&#163;13.55'
>>> "£13.55".encode('ascii', errors='backslashreplace')
b'\\xa313.55'
```

Décodage

```
>>> b = "£13.55".encode('utf8')
>>> b.decode('ascii', errors='replace')
'��13.55'
>>> b.decode('ascii', errors='ignore')
'13.55'
>>> b.decode('ascii', errors='backslashreplace')
'\\xc2\\xa313.55'
```

Moral

Il ressort clairement de ce qui précède qu'il est essentiel de garder vos encodages corrects lorsque vous utilisez unicode et des octets.

Fichier I / O

Les fichiers ouverts en mode non binaire (par exemple, `'r'` ou `'w'`) traitent les chaînes. Le codage sourd est `'utf8'` .

```
open(fn, mode='r')                    # opens file for reading in utf8
open(fn, mode='r', encoding='utf16')  # opens file for reading utf16

# ERROR: cannot write bytes when a string is expected:
open("foo.txt", "w").write(b"foo")
```

Les fichiers ouverts en mode binaire (par exemple `'rb'` ou `'wb'`) traitent les octets. Aucun

argument de codage ne peut être spécifié car il n'y a pas de codage.

```
open(fn, mode='wb')  # open file for writing bytes

# ERROR: cannot write string when bytes is expected:
open(fn, mode='wb').write("hi")
```

Lire Unicode et octets en ligne: https://riptutorial.com/fr/python/topic/1216/unicode-et-octets

Chapitre 199: urllib

Examples

HTTP GET

Python 2.x 2.7

Python 2

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

Utiliser `urllib.urlopen()` renverra un objet de réponse, qui peut être traité comme un fichier.

```
print response.code
# Prints: 200
```

Le `response.code` représente la valeur de retour http. 200 est OK, 404 est NotFound, etc.

```
print response.read()
'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack. etc'
```

`response.read()` et `response.readlines()` peuvent être utilisés pour lire le fichier HTML réel renvoyé par la requête. Ces méthodes fonctionnent de manière similaire à `file.read*`

Python 3.x 3.0

Python 3

```
import urllib.request

print(urllib.request.urlopen("http://stackoverflow.com/documentation/"))
# Prints: <http.client.HTTPResponse at 0x7f37a97e3b00>

response = urllib.request.urlopen("http://stackoverflow.com/documentation/")

print(response.code)
# Prints: 200
print(response.read())
# Prints: b'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack
Overflow</title>
```

Le module a été mis à jour pour Python 3.x, mais les cas d'utilisation restent fondamentalement les mêmes. `urllib.request.urlopen` retournera un objet similaire à un fichier.

HTTP POST

Pour que les données POST transmettent les arguments de requête codés en tant que données à urlopen ()

Python 2.x 2.7

Python 2

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.urlencode(query_parms)
response = urllib.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: '<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Log In - Stack Overflow'
```

Python 3.x 3.0

Python 3

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.parse.urlencode(query_parms).encode('utf-8')
response = urllib.request.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: b'<!DOCTYPE html>\r\n<html>....etc'
```

Décoder les octets reçus en fonction du codage du type de contenu

Les octets reçus doivent être décodés avec le codage de caractères correct pour être interprétés comme du texte:

Python 3.x 3.0

```
import urllib.request

response = urllib.request.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Python 2.x 2.7

```
import urllib2
response = urllib2.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().getencoding()
html = data.decode(encoding)
```

Lire urllib en ligne: https://riptutorial.com/fr/python/topic/2645/urllib

Chapitre 200: Utilisation du module "pip": PyPI Package Manager

Introduction

Parfois, vous devrez peut-être utiliser le gestionnaire de paquets pip dans python, par exemple. lorsque certaines importations peuvent générer `ImportError` et que vous souhaitez gérer l'exception. Si vous décompressez sur Windows, `Python_root/Scripts/pip.exe` intérieur `__main__.py` fichier `__main__.py` , où `main` classe `main` du paquetage `pip` est importée. Cela signifie que le paquetage pip est utilisé chaque fois que vous utilisez l'exécutable pip. Pour utiliser pip comme exécutable, voir: pip: PyPI Package Manager

Syntaxe

- pip. <function | attribute | class> où la fonction est l'une des suivantes:
  - autocomplete ()
    - Achèvement de la commande et de l'option pour l'analyseur principal (et les options) et ses sous-commandes (et options). Activer en recherchant l'un des scripts de complétion (bash, zsh ou fish).
  - check_isolated (args)
    - param args {list}
    - retourne {boolean}
  - create_main_parser ()
    - retourne l'objet {pip.baseparser.ConfigOptionParser}
  - main (args = None)
    - param args {list}
    - renvoie {entier} S'il n'a pas échoué, renvoie 0
  - parseopts (args)
    - param args {list}
  - get_installed_distributions ()
    - retourne {liste}
  - get_similar_commands (name)
    - Nom de la commande corrigé automatiquement
    - param name {string}
    - retourne {boolean}
  - get_summaries (commandé = True)
    - Donne les tuples triés (nom de la commande, résumé de la commande).
  - get_prog ()
    - renvoie {chaîne}
  - dist_is_editable (dist)
    - La distribution est-elle une installation modifiable?
    - param dist {object}
    - retourne {boolean}

- commands_dict
  - attribut {dictionary}

Examples

Exemple d'utilisation de commandes

```
import pip

command = 'install'
parameter = 'selenium'
second_param = 'numpy' # You can give as many package names as needed
switch = '--upgrade'

pip.main([command, parameter, second_param, switch])
```

Seuls les paramètres nécessaires sont obligatoires, donc `pip.main(['freeze'])` et `pip.main(['freeze', '', ''])` sont acceptables.

Installation par lots

Il est possible de transmettre de nombreux noms de paquet en un seul appel, mais si une installation / mise à niveau échoue, le processus d'installation complet s'arrête et se termine avec le statut '1'.

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

Si vous ne voulez pas vous arrêter lorsque certaines installations échouent, appelez l'installation en boucle.

```
for i in installed:
        pip.main(['install']+i.key+['--upgrade'])
```

Gestion des exceptions ImportError

Lorsque vous utilisez le fichier python en tant que module, il n'est pas toujours nécessaire de vérifier si le package est installé, mais il est toujours utile pour les scripts.

```
if __name__ == '__main__':
    try:
        import requests
    except ImportError:
        print("To use this module you need 'requests' module")
        t = input('Install requests? y/n: ')
```

```
        if t == 'y':
            import pip
            pip.main(['install', 'requests'])
            import requests
            import os
            import sys
            pass
        else:
            import os
            import sys
            print('Some functionality can be unavailable.')
else:
    import requests
    import os
    import sys
```

Installer force

Par exemple, de nombreux paquets sur la version 3.4 fonctionneraient sur la version 3.6, mais s'il n'y a pas de distributions pour une plate-forme spécifique, ils ne peuvent pas être installés, mais il existe une solution de contournement. Dans les fichiers .whl (appelés roues), la convention de nommage détermine si vous pouvez installer le package sur la plate-forme spécifiée. Par exemple. `scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl` [nom_package] - [version] - [interpréteur python] - [python-interpreter] - [système d'exploitation] .whl. Si le nom du fichier de la roue est modifié, de sorte que la plate-forme corresponde, pip essaie d'installer le paquet même si la version de la plateforme ou de python ne correspond pas. La suppression de la plate-forme ou de l'interpréteur du nom entraînera une erreur dans la dernière version du module pip `kjhfkjdf.whl is not a valid wheel filename.` .

Alternativement, le fichier .whl peut être décompressé en utilisant un archiveur comme 7-zip. - Il contient généralement un dossier de méta de distribution et un dossier avec les fichiers source. Ces fichiers source peuvent être simplement décompressés dans le répertoire `site-packges` , à moins que cette roue ne contienne un script d'installation, si c'est le cas, elle doit être exécutée en premier.

Lire Utilisation du module "pip": PyPI Package Manager en ligne: https://riptutorial.com/fr/python/topic/10730/utilisation-du-module--pip---pypi-package-manager

Chapitre 201: Utiliser des boucles dans les fonctions

Introduction

Dans Python, la fonction sera renvoyée dès que l'exécution aura atteint l'instruction "return".

Examples

Déclaration de retour dans la boucle dans une fonction

Dans cet exemple, la fonction retournera dès que la valeur var aura 1

```
def func(params):
    for value in params:
        print ('Got value {}'.format(value))

        if value == 1:
            # Returns from function as soon as value is 1
            print (">>>> Got 1")
            return

        print ("Still looping")

    return "Couldn't find 1"

func([5, 3, 1, 2, 8, 9])
```

sortie

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>>> Got 1
```

Lire Utiliser des boucles dans les fonctions en ligne:
https://riptutorial.com/fr/python/topic/10883/utiliser-des-boucles-dans-les-fonctions

Chapitre 202: Vérification de l'existence du chemin et des autorisations

Paramètres

Paramètre	Détails
os.F_OK	Valeur à transmettre en tant que paramètre de mode d'accès () pour tester l'existence du chemin.
os.R_OK	Valeur à inclure dans le paramètre mode de access () pour tester la lisibilité du chemin.
os.W_OK	Valeur à inclure dans le paramètre mode de access () pour tester la possibilité d'écriture du chemin.
os.X_OK	Valeur à inclure dans le paramètre mode de access () pour déterminer si le chemin peut être exécuté.

Examples

Effectuer des vérifications avec os.access

`os.access` est une solution bien meilleure pour vérifier si le répertoire existe et s'il est accessible en lecture et en écriture.

```
import os
path = "/home/myFiles/directory1"

## Check if path exists
os.access(path, os.F_OK)

## Check if path is Readable
os.access(path, os.R_OK)

## Check if path is Wriable
os.access(path, os.W_OK)

## Check if path is Execuatble
os.access(path, os.E_OK)
```

il est également possible de passer tous les contrôles ensemble

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.E_OK)
```

Tout ce qui précède renvoie `True` si l'accès est autorisé et `False` si non autorisé. Ceux-ci sont

disponibles sur Unix et Windows.

Lire Vérification de l'existence du chemin et des autorisations en ligne: https://riptutorial.com/fr/python/topic/1262/verification-de-l-existence-du-chemin-et-des-autorisations

Chapitre 203: Visualisation de données avec Python

Examples

Matplotlib

Matplotlib est une bibliothèque de traçage mathématique pour Python qui fournit une variété de fonctionnalités de traçage différentes.

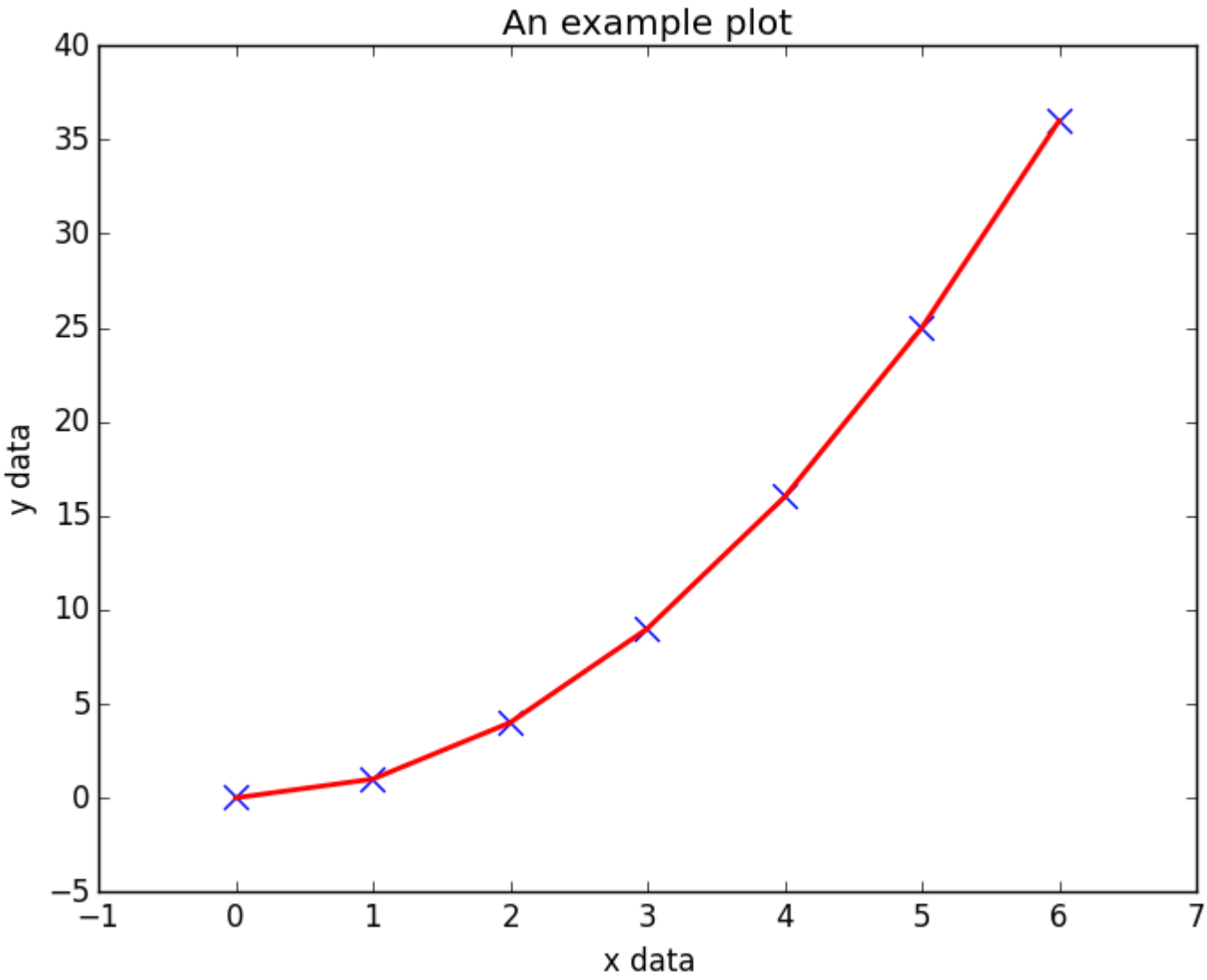
La documentation de matplotlib peut être trouvée ici , les SO Docs étant disponibles ici .

Matplotlib fournit deux méthodes distinctes pour le traçage, bien qu'elles soient interchangeables pour la plupart:

- Tout d'abord, matplotlib fournit l'interface `pyplot` interface directe et simple à utiliser qui permet de tracer des graphiques complexes dans un style similaire à MATLAB.
- Deuxièmement, matplotlib permet à l'utilisateur de contrôler directement les différents aspects (axes, lignes, ticks, etc.) en utilisant un système basé sur des objets. Ceci est plus difficile mais permet un contrôle complet sur toute la parcelle.

Voici un exemple d'utilisation de l'interface `pyplot` pour tracer des données générées:

```
import matplotlib.pyplot as plt

# Generate some data for plotting.
x = [0, 1, 2, 3, 4, 5, 6]
y = [i**2 for i in x]

# Plot the data x, y with some keyword arguments that control the plot style.
# Use two different plot commands to plot both points (scatter) and a line (plot).

plt.scatter(x, y, c='blue', marker='x', s=100) # Create blue markers of shape "x" and size 100
plt.plot(x, y, color='red', linewidth=2) # Create a red line with linewidth 2.

# Add some text to the axes and a title.
plt.xlabel('x data')
plt.ylabel('y data')
plt.title('An example plot')

# Generate the plot and show to the user.
plt.show()
```



Notez que `plt.show()` est connu pour être problématique dans certains environnements en raison de l'exécution de `matplotlib.pyplot` en mode interactif, et si tel est le cas, le comportement de blocage peut être explicitement remplacé par un argument facultatif, `plt.show(block=True)` , pour atténuer le problème.

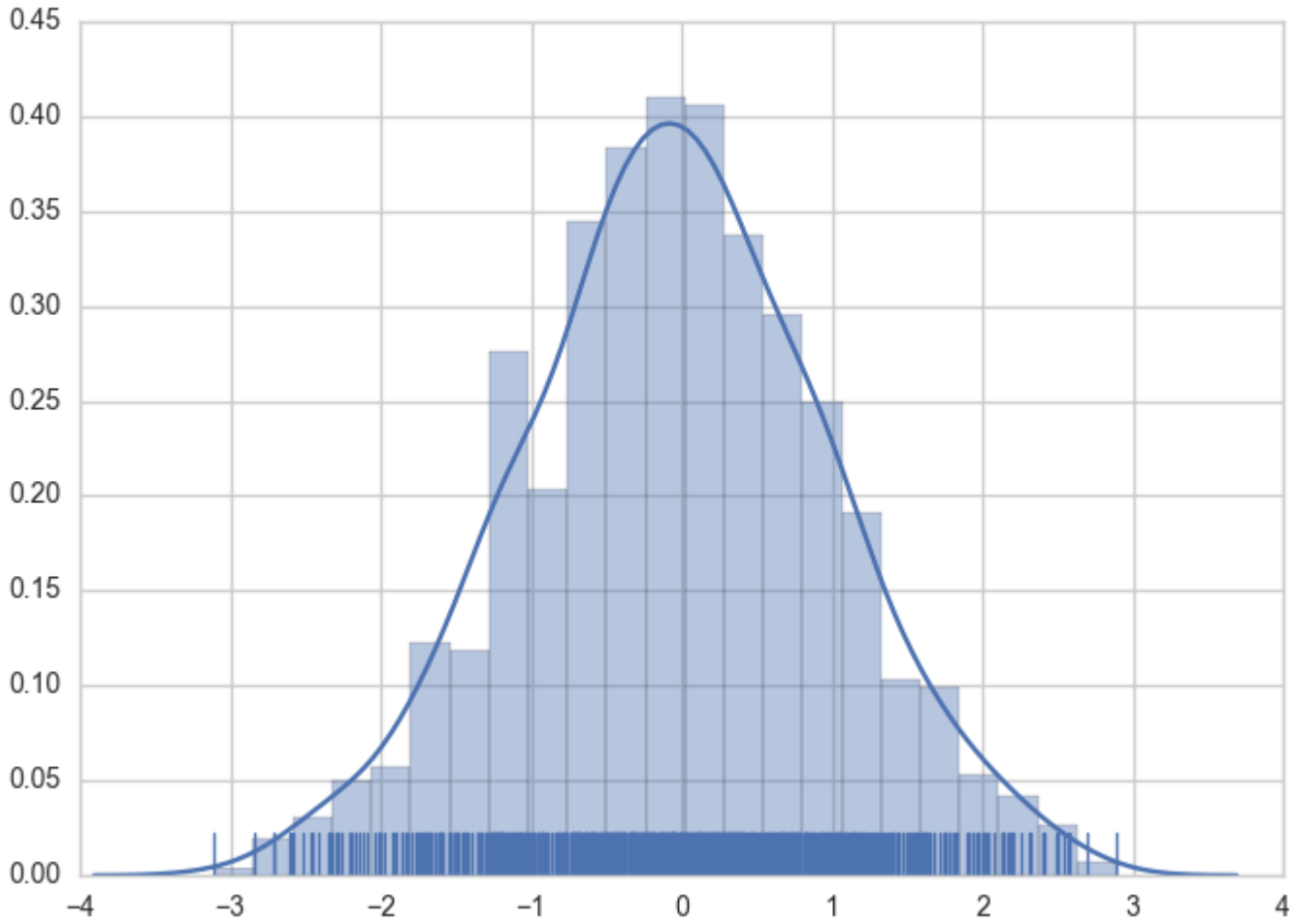
Seaborn

Seaborn est une enveloppe autour de Matplotlib qui facilite la création de tracés statistiques communs. La liste des tracés pris en charge comprend des tracés de distribution univariés et bivariés, des tracés de régression et un certain nombre de méthodes pour tracer des variables catégorielles. La liste complète des parcelles fournies par Seaborn est dans leur référence API .

Créer des graphiques dans Seaborn est aussi simple que d'appeler la fonction graphique appropriée. Voici un exemple de création d'un histogramme, d'une estimation de la densité du noyau et d'un tracé pour les données générées de manière aléatoire.

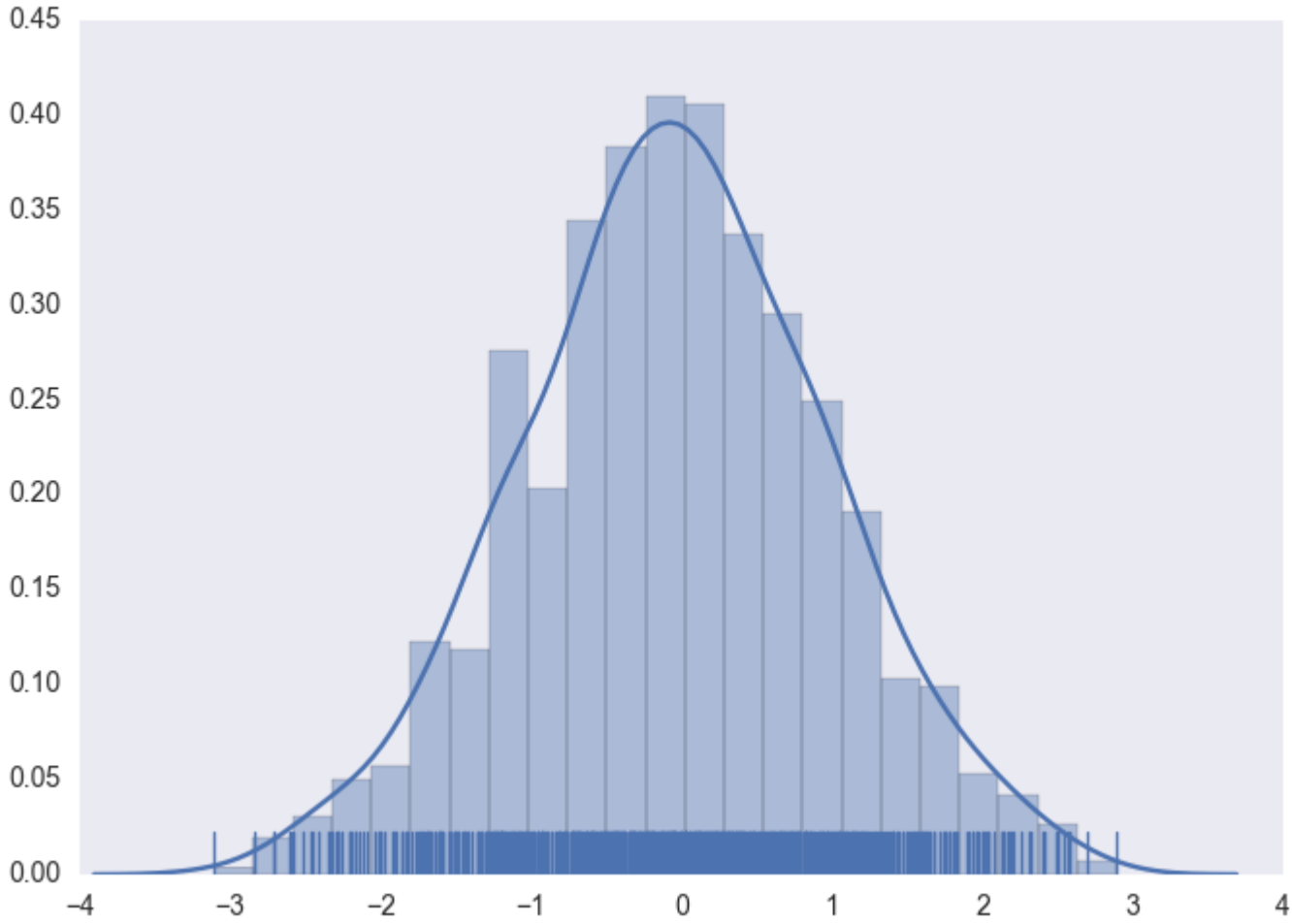
```
import numpy as np  # numpy used to create data from plotting
import seaborn as sns  # common form of importing seaborn
```

```
# Generate normally distributed data
data = np.random.randn(1000)

# Plot a histogram with both a rugplot and kde graph superimposed
sns.distplot(data, kde=True, rug=True)
```



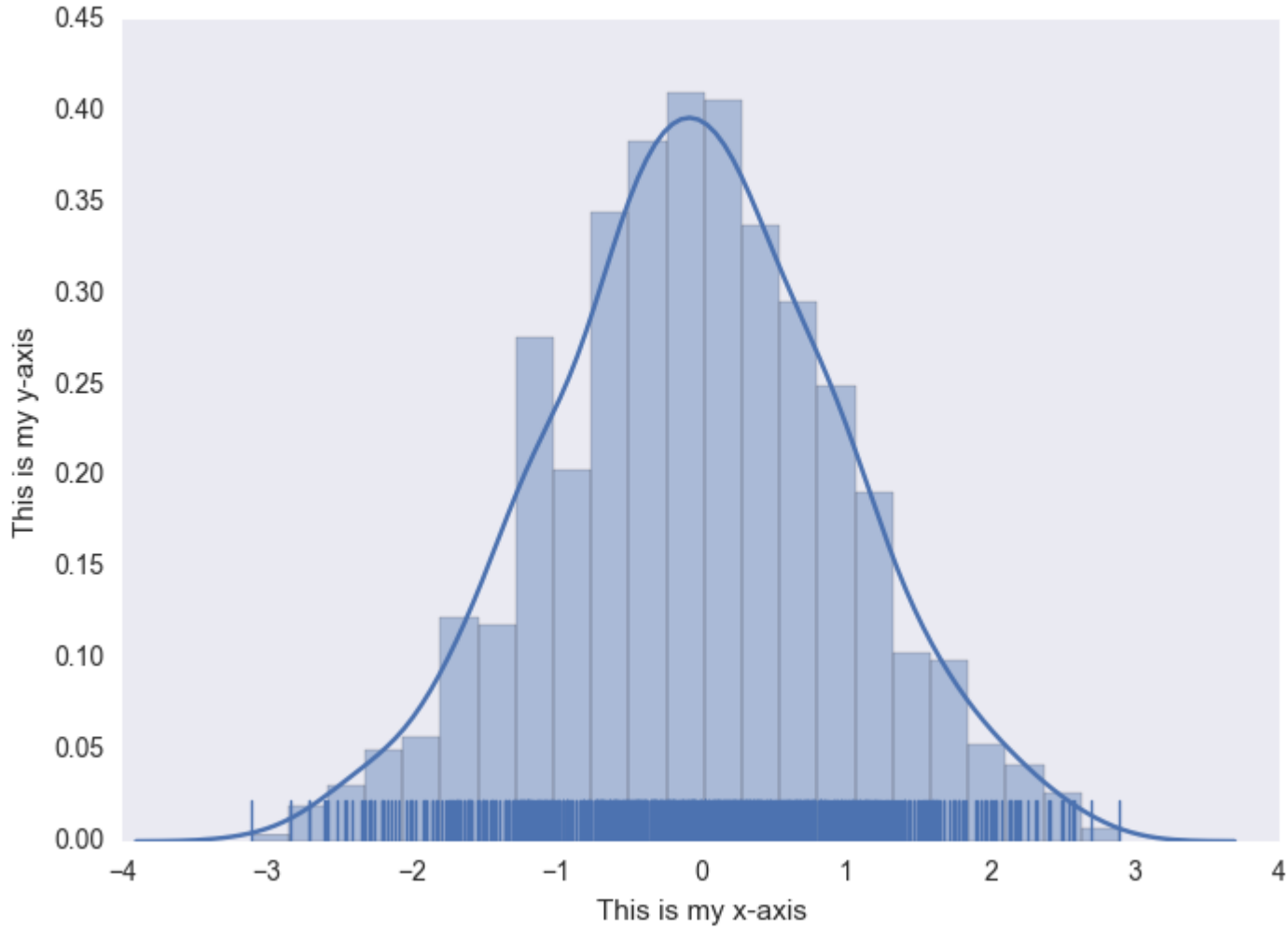
Le style de l'intrigue peut également être contrôlé en utilisant une syntaxe déclarative.

```
# Using previously created imports and data.

# Use a dark background with no grid.
sns.set_style('dark')
# Create the plot again
sns.distplot(data, kde=True, rug=True)
```



En prime, les commandes normales de matplotlib peuvent toujours être appliquées aux tracés de Seaborn. Voici un exemple d'ajout de titres d'axes à notre histogramme précédemment créé.

```
# Using previously created data and style

# Access to matplotlib commands
import matplotlib.pyplot as plt

# Previously created plot.
sns.distplot(data, kde=True, rug=True)
# Set the axis labels.
plt.xlabel('This is my x-axis')
plt.ylabel('This is my y-axis')
```



MayaVI

MayaVI est un outil de visualisation 3D pour les données scientifiques. Il utilise le Visualization Tool Kit ou VTK sous le capot. Utilisant la puissance de VTK , **MayaVI** est capable de produire une variété de tracés et de figures en trois dimensions. Il est disponible sous forme de logiciel séparé et de bibliothèque. Semblable à Matplotlib , cette bibliothèque fournit une interface de langage de programmation orientée objet pour créer des tracés sans avoir à connaître **VTK** .

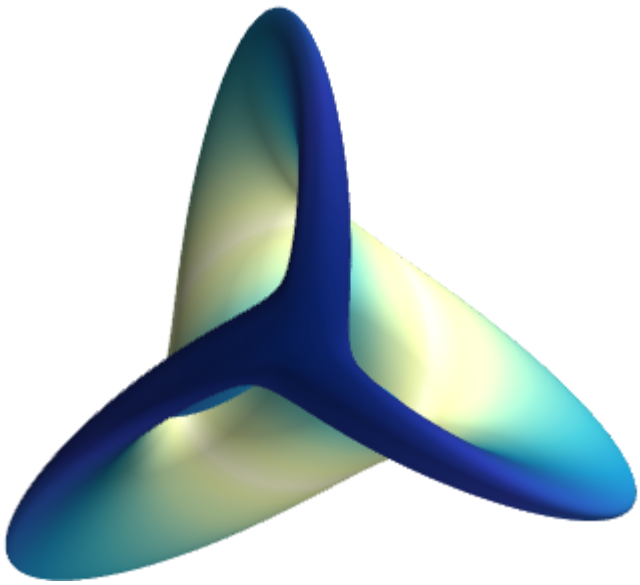
MayaVI est disponible uniquement dans les séries Python 2.7x! On espère être bientôt disponible dans la série Python 3-x! (Bien que certains succès soient constatés lors de l'utilisation de ses dépendances dans Python 3)

La documentation peut être trouvée ici . Quelques exemples de galeries se trouvent ici

Voici un exemple de tracé créé à l'aide de **MayaVI** dans la documentation.

```
# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>
# Copyright (c) 2007, Enthought, Inc.
# License: BSD Style.
```

```
from numpy import sin, cos, mgrid, pi, sqrt
from mayavi import mlab

mlab.figure(fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))
u, v = mgrid[- 0.035:pi:0.01, - 0.035:pi:0.01]

X = 2 / 3. * (cos(u) * cos(2 * v)
        + sqrt(2) * sin(u) * cos(v)) * cos(u) / (sqrt(2) -
                                                 sin(2 * u) * sin(3 * v))
Y = 2 / 3. * (cos(u) * sin(2 * v) -
        sqrt(2) * sin(u) * sin(v)) * cos(u) / (sqrt(2)
        - sin(2 * u) * sin(3 * v))
Z = -sqrt(2) * cos(u) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
S = sin(u)

mlab.mesh(X, Y, Z, scalars=S, colormap='YlGnBu', )

# Nice view from the front
mlab.view(.0, - 5.0, 4)
mlab.show()
```



Plotly

Plotly est une plate-forme moderne de traçage et de visualisation de données. Utile pour produire une variété de tracés, en particulier pour les sciences des données, **Plotly** est disponible sous forme de bibliothèque pour **Python** , **R** , **JavaScript** , **Julia** et **MATLAB** . Il peut également être utilisé comme application Web avec ces langues.

Les utilisateurs peuvent installer la bibliothèque de traçage et l'utiliser hors ligne après l'authentification de l'utilisateur. L'installation de cette bibliothèque et son authentification hors ligne sont données ici . En outre, les parcelles peuvent également être réalisées dans les **cahiers Jupyter** .

L'utilisation de cette bibliothèque nécessite un compte avec un nom d'utilisateur et un mot de passe. Cela donne à l'espace de travail pour enregistrer des tracés et des données sur le cloud.

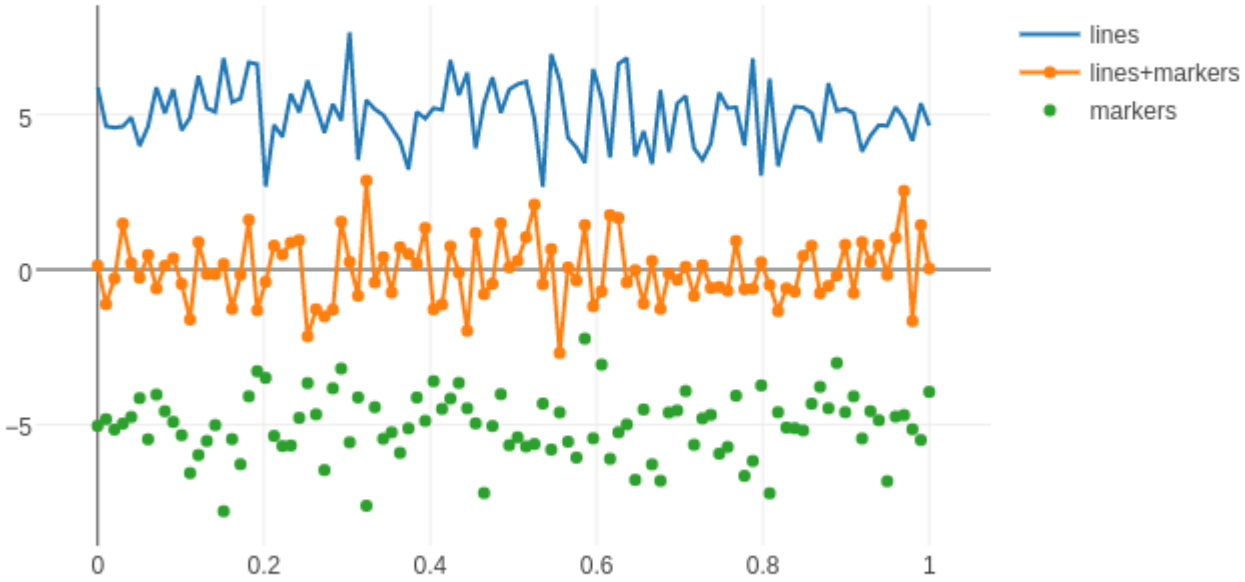
La version gratuite de la bibliothèque comporte des fonctionnalités légèrement limitées et conçue

pour créer 250 parcelles par jour. La version payante a toutes les fonctionnalités, les téléchargements illimités de tracés et plus de stockage de données privées. Pour plus de détails, on peut visiter la page principale ici .

Pour la documentation et des exemples, on peut aller ici

Un exemple de tracé à partir des exemples de documentation:

```
import plotly.graph_objs as go
import plotly as ply

# Create random data with numpy
import numpy as np

N = 100
random_x = np.linspace(0, 1, N)
random_y0 = np.random.randn(N)+5
random_y1 = np.random.randn(N)
random_y2 = np.random.randn(N)-5

# Create traces
trace0 = go.Scatter(
    x = random_x,
y = random_y0,
mode = 'lines',
name = 'lines'
)
trace1 = go.Scatter(
    x = random_x,
    y = random_y1,
    mode = 'lines+markers',
    name = 'lines+markers'
)
trace2 = go.Scatter(
    x = random_x,
    y = random_y2,
    mode = 'markers',
    name = 'markers'
)
data = [trace0, trace1, trace2]

ply.offline.plot(data, filename='line-mode')
```



Lire Visualisation de données avec Python en ligne:
https://riptutorial.com/fr/python/topic/2388/visualisation-de-donnees-avec-python

Chapitre 204: Vitesse du programme Python

Examples

Notation

Idée basique

La notation utilisée pour décrire la vitesse de votre programme Python s'appelle la notation Big-O. Disons que vous avez une fonction:

```
def list_check(to_check, the_list):
    for item in the_list:
        if to_check == item:
          return True
    return False
```

Ceci est une fonction simple pour vérifier si un élément est dans une liste. Pour décrire la complexité de cette fonction, vous allez dire O (n). Cela signifie "Ordre de n" car la fonction O est appelée fonction Ordre.

O (n) - généralement n le nombre d'éléments dans le conteneur

O (k) - généralement k est la valeur du paramètre ou le nombre d'éléments du paramètre

Opérations de liste

Opérations: Cas moyen (suppose que les paramètres sont générés aléatoirement)

Append: O (1)

Copie: O (n)

Del slice: O (n)

Supprimer l'article: O (n)

Insérer: O (n)

Obtenir l'article: O (1)

Set item: O (1)

Itération: O (n)

Obtenir une tranche: O (k)

Définir la tranche: O (n + k)

Étendre: O (k)

Trier: O (n log n)

Multiplier: O (nk)

x dans s: O (n)

min (s), max (s): O (n)

Longueur: O (1)

Opérations de deque

Un deque est une file d'attente double.

```
class Deque:
def __init__(self):
    self.items = []

def isEmpty(self):
    return self.items == []

def addFront(self, item):
    self.items.append(item)

def addRear(self, item):
    self.items.insert(0,item)

def removeFront(self):
    return self.items.pop()

def removeRear(self):
    return self.items.pop(0)

def size(self):
    return len(self.items)
```

Opérations: Cas moyen (suppose que les paramètres sont générés aléatoirement)

Append: O (1)

Appendleft: O (1)

Copie: O (n)

Étendre: O (k)

Extendleft: O (k)

Pop: O (1)

Popleft: O (1)

Supprimer: O (n)

Rotation: O (k)

Définir les opérations

Operation: Average Case (suppose des paramètres générés aléatoirement): le pire des cas

x dans s: O (1)

Différence s - t: O (len (s))

Intersection s & t: O (min (len (s), len (t))): O (len (s) * len (t)

Intersection multiple s1 & s2 & s3 & ... & sn:: (n-1) * O (l) où l est max (len (s1), ..., len (sn))

s.difference_update (t): O (len (t)): O (len (t) * len (s))

s.symetric_difference_update (t): O (len (t))

Différence symétrique s ^ t: O (len (s)): O (len (s) * len (t))

Union s | t: O (len (s) + len (t))

Notations algorithmiques ...

Certains principes s'appliquent à l'optimisation dans tout langage informatique, et Python ne fait pas exception. **N'optimisez pas au fur et à mesure** : écrivez votre programme sans vous soucier des optimisations possibles, en vous concentrant plutôt sur le fait que le code est propre, correct et compréhensible. Si c'est trop gros ou trop lent lorsque vous avez terminé, alors vous pouvez envisager de l'optimiser.

Rappelez-vous la règle des 80/20 : dans de nombreux domaines, vous pouvez obtenir 80% du résultat avec 20% de l'effort (également appelé la règle 90/10 - cela dépend de qui vous parlez). Chaque fois que vous êtes sur le point d'optimiser le code, utilisez le profilage pour savoir où se situe 80% du temps d'exécution, afin de savoir où concentrer vos efforts.

Exécutez toujours des tests "avant" et "après" : comment saurez-vous que vos optimisations ont fait une différence? Si votre code optimisé s'avère être légèrement plus rapide ou plus petit que la version d'origine, annulez vos modifications et revenez au code d'origine.

Utilisez les bons algorithmes et structures de données: N'utilisez pas un algorithme de tri par bulles O (n2) pour trier un millier d'éléments lorsqu'un test rapide O (n log n) est disponible. De même, ne stockez pas un millier d'éléments dans un tableau qui nécessite une recherche O (n) lorsque vous pouvez utiliser un arbre binaire O (log n) ou une table de hachage Python O (1).

Pour plus de détails, visitez le lien ci-dessous ... Python Speed Up

Les trois notations asymptotiques suivantes sont principalement utilisées pour représenter la complexité temporelle des algorithmes.

1. **Ation Notation** : La notation thêta délimite une fonction de haut en bas, elle définit donc un

comportement asymptotique exact. Un moyen simple d'obtenir la notation Theta d'une expression consiste à supprimer les termes d'ordre faible et à ignorer les constantes principales. Par exemple, considérons l'expression suivante. 3n3 + 6n2 + 6000 = Θ (n3) L'abandon des termes d'ordre inférieur est toujours correct car il y aura toujours un n0 après lequel Θ (n3) aura des valeurs supérieures à Θn2) quelles que soient les constantes impliquées. Pour une fonction donnée g (n), on note Θ (g (n)) est le jeu de fonctions suivant. Θ (g (n)) = {f (n): il existe des constantes positives c1, c2 et n0 telles que 0 <= c1 *g (n)* <= *f (n)* <= *c2* g (n) pour tout n> = n0} La définition ci-dessus signifie que si f (n) est thêta de g (n), alors la valeur f (n) est toujours comprise entre c1 *g (n) et c2* g (n) pour les grandes valeurs de n (n> = n0). La définition de thêta exige également que f (n) soit non négatif pour les valeurs de n supérieures à n0.

2. **Notation Big O** : La notation Big O définit une limite supérieure d'un algorithme, elle limite une fonction uniquement par le haut. Par exemple, considérons le cas du tri par insertion. Dans le meilleur des cas, cela prend du temps linéaire dans le meilleur des cas et du temps quadratique. On peut dire sans se tromper que la complexité temporelle du tri par insertion est O (n ^ 2). Notez que O (n ^ 2) couvre également le temps linéaire. Si nous utilisons la notation Θ pour représenter la complexité temporelle du tri par insertion, nous devons utiliser deux instructions pour le meilleur et le pire des cas:

1. La pire complexité temporelle du tri par insertion est Θ (n ^ 2).
2. La meilleure complexité temporelle du tri par insertion est Θ (n).

La notation Big O est utile lorsque la complexité temporelle d'un algorithme est uniquement liée à la limite supérieure. Souvent, nous trouvons facilement une limite supérieure en regardant simplement l'algorithme. O (g (n)) = {f (n): il existe des constantes positives c et n0 telles que 0 <= f (n) <= cg (n) pour tout n> = n0}

3. **Notation Ω** : Tout comme la notation Big O fournit une borne supérieure asymptotique sur une fonction, la notation Ω fournit une borne inférieure asymptotique. Ω Notation <peut être utile lorsque la complexité temporelle d'un algorithme est inférieure. Comme discuté dans le post précédent, la meilleure performance d'un algorithme n'est généralement pas utile, la notation Omega est la notation la moins utilisée parmi les trois. Pour une fonction donnée g (n), on note Ω (g (n)) l'ensemble des fonctions. Ω (g (n)) = {f (n): il existe des constantes positives c et n0 telles que 0 <= cg (n) <= f (n) pour tout n> = n0}. Considérons ici le même exemple de tri par insertion. La complexité temporelle du tri par insertion peut être écrite comme Ω (n), mais ce n'est pas une information très utile sur le tri par insertion, car nous sommes généralement intéressés par le pire des cas et parfois le cas moyen.

Lire Vitesse du programme Python en ligne: https://riptutorial.com/fr/python/topic/9185/vitesse-du-programme-python

Chapitre 205: Web grattant avec Python

Introduction

Le Web scraping est un processus automatisé et programmé grâce auquel les données peuvent être constamment «grattées» sur les pages Web. Également connu sous le nom de récupération d'écran ou de collecte Web, le balayage Web peut fournir des données instantanées à partir de n'importe quelle page Web accessible au public. Sur certains sites Web, le raclage Web peut être illégal.

Remarques

Paquets Python utiles pour le web scraping (ordre alphabétique)

Faire des demandes et collecter des données

`requests`

Un package simple mais puissant pour faire des requêtes HTTP.

`requests-cache`

Mise en cache pour les `requests` ; la mise en cache des données est très utile. En développement, cela signifie que vous pouvez éviter de frapper un site inutilement. Lorsque vous exécutez une collection réelle, cela signifie que si votre racloir plante pour une raison quelconque (vous n'avez peut-être pas manipulé de contenu inhabituel sur le site ...? Peut-être que le site est tombé ...?) D'où tu t'es arrêté.

`scrapy`

Utile pour créer des robots d'indexation sur le Web, où vous avez besoin de quelque chose de plus puissant que l'utilisation de `requests` et l'itération de pages.

`selenium`

Liaisons Python pour Selenium WebDriver, pour l'automatisation des navigateurs. L'utilisation de `requests` pour effectuer directement des requêtes HTTP est souvent plus simple pour récupérer des pages Web. Cependant, cela reste un outil utile lorsqu'il n'est pas possible de reproduire le comportement souhaité d'un site à l'aide de `requests` uniquement, en particulier lorsque JavaScript est requis pour rendre des éléments sur une page.

Analyse HTML

BeautifulSoup

Requête de documents HTML et XML à l'aide de plusieurs analyseurs (analyseur HTML intégré à Python, `html5lib` , `lxml` ou `lxml.html`)

lxml

Traite HTML et XML. Peut être utilisé pour interroger et sélectionner du contenu à partir de documents HTML via des sélecteurs CSS et XPath.

Examples

Exemple de base d'utilisation de requêtes et de lxml pour récupérer des données

```
# For Python 2 compatibility.
from __future__ import print_function

import lxml.html
import requests


def main():
    r = requests.get("https://httpbin.org")
    html_source = r.text
    root_element = lxml.html.fromstring(html_source)
    # Note root_element.xpath() gives a *list* of results.
    # XPath specifies a path to the element we want.
    page_title = root_element.xpath('/html/head/title/text()')[0]
    print(page_title)

if __name__ == '__main__':
    main()
```

Maintenir la session Web-scraping avec les requêtes

Il est recommandé de conserver une session Web pour conserver les cookies et autres paramètres. En outre, cela peut entraîner une *amélioration des performances,* car `requests.Session` réutilise la connexion TCP sous-jacente à un hôte:

```
import requests

with requests.Session() as session:
    # all requests through session now have User-Agent header set
    session.headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36'}

    # set cookies
    session.get('http://httpbin.org/cookies/set?key=value')

    # get cookies
    response = session.get('http://httpbin.org/cookies')
    print(response.text)
```

Gratter en utilisant le cadre Scrapy

Vous devez d'abord créer un nouveau projet Scrapy. Entrez un répertoire dans lequel vous souhaitez stocker votre code et exécutez:

```
scrapy startproject projectName
```

Pour gratter, nous avons besoin d'une araignée. Les araignées définissent la manière dont un site donné sera raclé. Voici le code d'une araignée qui suit les liens vers les questions les plus votées sur StackOverflow et raconte certaines données de chaque page (source):

```
import scrapy

class StackOverflowSpider(scrapy.Spider):
    name = 'stackoverflow'  # each spider has a unique name
    start_urls = ['http://stackoverflow.com/questions?sort=votes']  # the parsing starts from
a specific set of urls

    def parse(self, response):  # for each request this generator yields, its response is sent
to parse_question
        for href in response.css('.question-summary h3 a::attr(href)'):  # do some scraping
stuff using css selectors to find question urls
            full_url = response.urljoin(href.extract())
            yield scrapy.Request(full_url, callback=self.parse_question)

    def parse_question(self, response):
        yield {
            'title': response.css('h1 a::text').extract_first(),
            'votes': response.css('.question .vote-count-post::text').extract_first(),
            'body': response.css('.question .post-text').extract_first(),
            'tags': response.css('.question .post-tag::text').extract(),
            'link': response.url,
        }
```

Enregistrez vos classes d'araignées dans le répertoire `projectName\spiders` . Dans ce cas - nom du `projectName\spiders\stackoverflow_spider.py` .

Vous pouvez maintenant utiliser votre araignée. Par exemple, essayez de courir (dans le répertoire du projet):

```
scrapy crawl stackoverflow
```

Modifier l'agent utilisateur Scrapy

Parfois, l'agent utilisateur Scrapy par défaut (`"Scrapy/VERSION (+http://scrapy.org)"`) est bloqué par l'hôte. Pour modifier l'agent utilisateur par défaut, ouvrez **settings.py** , décommentez et modifiez la ligne suivante selon vos souhaits.

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

Par exemple

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/51.0.2704.103 Safari/537.36'
```

Gratter à l'aide de BeautifulSoup4

```
from bs4 import BeautifulSoup
import requests

# Use the requests module to obtain a page
res = requests.get('https://www.codechef.com/problems/easy')

# Create a BeautifulSoup object
page = BeautifulSoup(res.text, 'lxml')   # the text field contains the source of the page

# Now use a CSS selector in order to get the table containing the list of problems
datatable_tags = page.select('table.dataTable')  # The problems are in the <table> tag,
                                                 # with class "dataTable"
# We extract the first tag from the list, since that's what we desire
datatable = datatable_tags[0]
# Now since we want problem names, they are contained in <b> tags, which are
# directly nested under <a> tags
prob_tags = datatable.select('a > b')
prob_names = [tag.getText().strip() for tag in prob_tags]

print prob_names
```

Scraping utilisant Selenium WebDriver

Certains sites Web n'aiment pas être effacés. Dans ces cas, vous devrez peut-être simuler un utilisateur réel travaillant avec un navigateur. Selenium lance et contrôle un navigateur Web.

```
from selenium import webdriver

browser = webdriver.Firefox()  # launch firefox browser

browser.get('http://stackoverflow.com/questions?sort=votes')  # load url

title = browser.find_element_by_css_selector('h1').text  # page title (first h1 element)

questions = browser.find_elements_by_css_selector('.question-summary')  # question list

for question in questions:  # iterate over questions
    question_title = question.find_element_by_css_selector('.summary h3 a').text
    question_excerpt = question.find_element_by_css_selector('.summary .excerpt').text
    question_vote = question.find_element_by_css_selector('.stats .vote .votes .vote-count-
post').text

    print "%s\n%s\n%s votes\n-----------\n" % (question_title, question_excerpt,
question_vote)
```

Le sélénium peut faire beaucoup plus. Il peut modifier les cookies du navigateur, remplir des formulaires, simuler des clics de souris, réaliser des captures d'écran de pages Web et exécuter du code JavaScript personnalisé.

Téléchargement de contenu Web simple avec urllib.request

Le module de bibliothèque standard `urllib.request` peut être utilisé pour télécharger du contenu Web:

```
from urllib.request import urlopen

response = urlopen('http://stackoverflow.com/questions?sort=votes')
data = response.read()

# The received bytes should usually be decoded according the response's character set
encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Un module similaire est également disponible dans Python 2 .

Grattage avec boucle

importations:

```
from subprocess import Popen, PIPE
from lxml import etree
from io import StringIO
```

Téléchargement:

```
user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.95 Safari/537.36'
url = 'http://stackoverflow.com'
get = Popen(['curl', '-s', '-A', user_agent, url], stdout=PIPE)
result = get.stdout.read().decode('utf8')
```

`-s` : téléchargement silencieux

`-A` : drapeau de l'agent utilisateur

Analyse:

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

Lire Web grattant avec Python en ligne: https://riptutorial.com/fr/python/topic/1792/web-grattant-avec-python

Chapitre 206: Websockets

Examples

Simple Echo avec aiohttp

`aiohttp` fournit des `aiohttp` asynchrones.

Python 3.x 3.5

```
import asyncio
from aiohttp import ClientSession

with ClientSession() as session:
    async def hello_world():

        websocket = await session.ws_connect("wss://echo.websocket.org")

        websocket.send_str("Hello, world!")

        print("Received:", (await websocket.receive()).data)

        await websocket.close()

    loop = asyncio.get_event_loop()
    loop.run_until_complete(hello_world())
```

Classe d'emballage avec aiohttp

`aiohttp.ClientSession` peut être utilisé comme parent pour une classe WebSocket personnalisée.

Python 3.x 3.5

```
import asyncio
from aiohttp import ClientSession

class EchoWebSocket(ClientSession):

    URL = "wss://echo.websocket.org"

    def __init__(self):
        super().__init__()
        self.websocket = None

    async def connect(self):
        """Connect to the WebSocket."""
        self.websocket = await self.ws_connect(self.URL)

    async def send(self, message):
        """Send a message to the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        self.websocket.send_str(message)
        print("Sent:", message)
```

```
    async def receive(self):
        """Receive one message from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        return (await self.websocket.receive()).data

    async def read(self):
        """Read messages from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"

        while self.websocket.receive():
            message = await self.receive()
            print("Received:", message)
            if message == "Echo 9!":
                break

async def send(websocket):
    for n in range(10):
        await websocket.send("Echo {}!".format(n))
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()

with EchoWebSocket() as websocket:

    loop.run_until_complete(websocket.connect())

    tasks = (
        send(websocket),
        websocket.read()
    )

    loop.run_until_complete(asyncio.wait(tasks))

    loop.close()
```

Utiliser Autobahn comme une usine Websocket

Le package Autobahn peut être utilisé pour les fabriques de serveurs Web socket Python.

Documentation du package Python Autobahn

Pour installer, il suffit généralement d'utiliser la commande terminal

(Pour Linux):

```
sudo pip install autobahn
```

(Pour les fenêtres):

```
python -m pip install autobahn
```

Ensuite, un serveur d'écho simple peut être créé dans un script Python:

```
from autobahn.asyncio.websocket import WebSocketServerProtocol
class MyServerProtocol(WebSocketServerProtocol):
```

```
    '''When creating server protocol, the
    user defined class inheriting the
    WebSocketServerProtocol needs to override
    the onMessage, onConnect, et-c events for
    user specified functionality, these events
    define your server's protocol, in essence'''
    def onMessage(self,payload,isBinary):
        '''The onMessage routine is called
        when the server receives a message.
        It has the required arguments payload
        and the bool isBinary. The payload is the
        actual contents of the "message" and isBinary
        is simply a flag to let the user know that
        the payload contains binary data. I typically
        elsewise assume that the payload is a string.
        In this example, the payload is returned to sender verbatim.'''
        self.sendMessage(payload,isBinary)
if__name__=='__main__':
    try:
        importasyncio
    except ImportError:
        '''Trollius = 0.3 was renamed'''
        import trollius as asyncio
    from autobahn.asyncio.websocketimportWebSocketServerFactory
    factory=WebSocketServerFactory()
    '''Initialize the websocket factory, and set the protocol to the
    above defined protocol(the class that inherits from
    autobahn.asyncio.websocket.WebSocketServerProtocol)'''
    factory.protocol=MyServerProtocol
    '''This above line can be thought of as "binding" the methods
    onConnect, onMessage, et-c that were described in the MyServerProtocol class
    to the server, setting the servers functionality, ie, protocol'''
    loop=asyncio.get_event_loop()
    coro=loop.create_server(factory,'127.0.0.1',9000)
    server=loop.run_until_complete(coro)
    '''Run the server in an infinite loop'''
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.close()
        loop.close()
```

Dans cet exemple, un serveur est en cours de création sur l'hôte local (127.0.0.1) sur le port 9000. Il s'agit de l'adresse IP et du port d'écoute. Ce sont des informations importantes, car en utilisant cela, vous pouvez identifier l'adresse LAN et le port de votre ordinateur à partir de votre modem, bien que tous les routeurs que vous avez sur l'ordinateur. Ensuite, en utilisant Google pour étudier votre adresse IP WAN, vous pouvez concevoir votre site Web pour envoyer des messages WebSocket à votre IP WAN, sur le port 9000 (dans cet exemple).

Il est important de transférer votre modem vers l'arrière, ce qui signifie que si vous avez des routeurs connectés en guirlande au modem, entrez les paramètres de configuration du modem, le port du modem vers le routeur connecté, etc., jusqu'au routeur final de votre ordinateur. est connecté est d'avoir l'information reçue sur le port modem 9000 (dans cet exemple) qui lui est transféré.

Lire Websockets en ligne: https://riptutorial.com/fr/python/topic/4751/websockets

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le langage Python	A. Raza, Aaron Critchley, Abhishek Jain, AER, afeique, Akshay Kathpal, alejosocorro, Alessandro Trinca Tornidor, Alex Logan, ALinuxLover, Andrea, Andrii Abramov, Andy, Andy Hayden, angussidney, Ani Menon, Anthony Pham, Antoine Bolvy, Aquib Javed Khan, Ares, Arpit Solanki, B8vrede, Baaing Cow, baranskistad, Brian C, Bryan P, BSL-5, BusyAnt, Cbeb24404, ceruleus, ChaoticTwist, Charlie H, Chris Midgley, Christian Ternus, Claudiu, Clíodhna, CodenameLambda, cLDs EED, Community, Conrad.Dean, Daksh Gupta, Dania, Daniel Minnaar, Darth Shadow, Dartmouth, deeenes, Delgan, depperm, DevD, dodell, Douglas Starnes, duckman_1991, Eamon Charles, edawine, Elazar, eli-bd, Enrico Maria De Angelis, Erica, Erica, ericdwang, Erik Godard, EsmaeelE, Filip Haglund, Firix, fox, Franck Dernoncourt, Fred Barclay, Freddy, Gerard Roche, glS, GoatsWearHats, GThamizh, H. Pauwelyn, hardmooth, hayalci, hichris123, Ian, IanAuld, icesin, Igor Raush, Ilyas Mimouni, itsthejoker, J F, Jabba, jalanb, James, James Taylor, Jean-Francois T., jedwards, Jeffrey Lin, jfunez, JGreenwell, Jim Fasarakis Hilliard, jim opleydulven, jimsug, jmunsch, Johan Lundberg, John Donner, John Slegers, john400, jonrsharpe, Joseph True, JRodDynamite, jtbandes, Juan T, Kamran Mackey, Karan Chudasama, KerDam, Kevin Brown, Kiran Vemuri, kisanme, Lafexlos, Leon, Leszek Kicior, LostAvatar, Majid, manu, MANU, Mark Miller, Martijn Pieters, Mathias711, matsjoyce, Matt, Mattew Whitt, mdegis, Mechanic, Media, mertyildiran, metahost, Mike Driscoll, MikJR, Miljen Mikic, mnoronha, Morgoth, moshemeirelles, MSD, MSeifert, msohng, msw, muddyfish, Mukund B, Muntasir Alam, Nathan Arthur, Nathaniel Ford, Ned Batchelder, Ni., niyasc, noɥʎd ʎzɐɹƆ, numbermaniac, orvi, Panda, Patrick Haugh, Pavan Nath, Peter Masiar, PSN, PsyKzz, pylang, pzp, Qchmqs, Quill, Rahul Nair, Rakitić, Ram Grandhi, rfkortekaas, rick112358, Robotski, rrao, Ryan Hilbert, Sam Krygsheld, Sangeeth Sudheer, SashaZd, Selcuk, Severiano Jaramillo Quintanar, Shiven, Shoe, Shog9, Sigitas Mockus, Simplans, Slayther, stark, StuxCrystal, SuperBiasedMan, Sнаđoɯƒa, taylor swift, techydesigner, Tejus Prasad, TerryA, The_Curry_Man, TheGenie OfTruth, Timotheus.Kampik, tjohnson, Tom Barron, Tom de Geus, Tony Suffolk 66, tonyo, TPVasconcelos,

		user2314737, user2853437, user312016, Utsav T, vaichidrewar, vasili111, Vin, W.Wong, weewooquestionaire, Will, wintermute, Yogendra Sharma, Zach Janicki, Zags
2	* args et ** kwargs	cjds, Eric Zhang, ericmarkmartin, Geeklhem, J F, Jeff Hutchins , Jim Fasarakis Hilliard, JuanPablo, kdopen, loading..., Marlon Abeykoon, Mattew Whitt, Pasha, pcurry, PsyKzz, Scott Mermelstein, user2314737, Valentin Lorentz, Veedrac
3	Accéder au code source Python et au bytecode	muddyfish, StuxCrystal, user2314737
4	Accès à la base de données	Alessandro Trinca Tornidor, Antonio, bee-sting, cldseed, D. Alveno, John Y, LostAvatar, mbsingh, Michel Touw, qwertyuip9 , RamenChef, rrawat, Stephen Leppik, Stephen Nyamweya, sumitroy, user2314737, valeas, zweiterlinde
5	Accès aux attributs	Elazar, SashaZd, SuperBiasedMan
6	Alternatives à changer de déclaration à partir d'autres langues	davidism, J F, zmo, Валерий Павлов
7	Analyse des arguments de ligne de commande	amblina, Braiam, Claudiu, cledoux, Elazar, Gerard Roche, krato, loading..., Marco Pashkov, Or Duan, Pasha, RamenChef , rfkortekaas, Simplans, Thomas Gerot, Topperfalkon, zmo, zondo
8	Analyse HTML	alecxe, talhasch
9	Anti-Patterns Python	Alessandro Trinca Tornidor, Annonymous, eenblam, Mahmoud Hashemi, RamenChef, Stephen Leppik
10	Appelez Python depuis C #	Julij Jegorov
11	Arbre de syntaxe abstraite	Teepeemm
12	ArcPy	Midavalo, PolyGeo, Zhanping Shi
13	Augmenter les erreurs / exceptions personnalisées	naren
14	Ballon	Stephen Leppik, Thomas Gerot
15	Bibliothèque de sous-	Adam Matan, Andrew Schade, Brendan Abel, jfs, jmunsch,

	processus	Riccardo Petraglia
16	Blocs de code, cadres d'exécution et espaces de noms	Jeremy, Mohammed Salman
17	Boucles	Adriano, Alex L, alfonso.kim, Alleo, Anthony Pham, Antti Haapala, Chris Hunt, Christian Ternus, Darth Kotik, DeepSpace, Delgan, DhiaTN, ebo, Elazar, Eric Finn, Felix D., Ffisegydd, Gal Dreiman, Generic Snake, ghostarbeiter, GoatsWearHats, Guy, Inbar Rose, intboolstring, J F, James, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, jrast, Karl Knechtel, machine yearning, Mahdi, manetsus, Martijn Pieters, Math, Mathias711, MSeifert, pnhgiol, rajah9, Rishabh Gupta, Ryan, sarvajeetsuman, sevenforce, SiggyF, Simplans, skrrgwasme, SuperBiasedMan, textshell, The_Curry_Man, Thomas Gerot, Tom, Tony Suffolk 66, user1349663, user2314737, Vinzee, Will
18	Calcul parallèle	Akshat Mahajan, Dair, Franck Dernoncourt, J F, Mahdi, nlsdfnbch, Ryan Smith, Vinzee, Xavier Combelle
19	Caractéristiques cachées	Aaron Hall, Akshat Mahajan, Anthony Pham, Antti Haapala, Byte Commander, dermen, Elazar, Ellis, ericmarkmartin, Fermi paradox, Ffisegydd, japborst, Jim Fasarakis Hilliard, jonrsharpe , Justin, kramer65, Lafexlos, LDP, Morgan Thrapp, muddyfish, nico, OrangeTux, pcurry, Pythonista, Selcuk, Serenity, Tejas Jadhav, tobias_k, Vlad Shcherbina, Will
20	ChemPy - package python	Biswa_9937
21	Classes d'extension et d'extension	2Cubed, proprefenetre, pylang, rrao, Simon Hibbs, Simplans
22	Classes de base abstraites (abc)	Akshat Mahajan, Alessandro Trinca Tornidor, JGreenwell, Kevin Brown, Mattew Whitt, mkrieger1, SashaZd, Stephen Leppik
23	Collecte des ordures	bogdanciobanu, Claudiu, Conrad.Dean, Elazar, FazeL, J F, James Elderfield, lukess, muddyfish, Sam Whited, SiggyF, Stephen Leppik, SuperBiasedMan, Xavier Combelle
24	commencer avec GZip	orvi
25	Commentaires et documentation	Ani Menon, FunkySayu, MattCorr, SuperBiasedMan, TuringTux

26	Communication série Python (pyserial)	Alessandro Trinca Tornidor, Ani Menon, girish946, mnoronha, Saranjith, user2314737
27	Comparaisons	Anthony Pham, Ares, Elazar, J F, MSeifert, Shawn Mehan, SuperBiasedMan, Will, Xavier Combelle
28	Compte	Andy Hayden, MSeifert, Peter Mølgaard Pallesen, pylang
29	Concurrence Python	David Heyman, Faiz Halde, Iván Rodríguez Torres, J F, Thomas Moreau, Tyler Gubala
30	Conditionnels	Andy Hayden, BusyAnt, Chris Larson, deepakkt, Delgan, Elazar, evuez, Ffisegydd, Geeklhem, Hannes Karppila, James, Kevin Brown, krato, Max Feng, noɥʎdʎzɐɔ, rajah9, rrao, SashaZd, Simplans, Slayther, Soumendra Kumar Sahoo, Thomas Gerot, Trimax, Valentin Lorentz, Vinzee, wwii, xgord, Zack
31	configparser	Chinmay Hegde, Dunatotatos
32	Connexion de Python à SQL Server	metmirr
33	Connexion sécurisée au shell en Python	mnoronha, Shijo
34	Copier des données	hashcode55, StuxCrystal
35	Cours de base avec Python	4444, Guy, kollery, Vinzee
36	Création d'un service Windows à l'aide de Python	Simon Hibbs
37	Créer des paquets Python	Claudiu, KeyWeeUsr, Marco Pashkov, pylang, SuperBiasedMan, Thtu
38	Créer un environnement virtuel avec virtualenvwrapper dans Windows	Sirajus Salayhin
39	ctypes	Or East
40	Date et l'heure	Ajean, alecxe, Andy, Antti Haapala, BusyAnt, Conrad.Dean, Elazar, ghostarbeiter, J F, Jeffrey Lin, jonrsharpe, Kevin Brown , Nicole White, nlsdfnbch, Ohad Eytan, Paul, paulmorriss, proprius, RahulHP, RamenChef, sagism, Simplans, Sirajus

		Salayhin, Suku, Will
41	Décorateurs	Alessandro Trinca Tornidor, ChaoticTwist, Community, Dair, doratheexplorer0911, Emolga, greut, iankit, JGreenwell, jonrsharpe, kefkius, Kevin Brown, Mattew Whitt, MSeifert, muddyfish, Mukunda Modell, Nearoo, Nemo, Nuno André, Pasha, Rob Bednark, seenu s, Shreyash S Sarnayak, Simplans, StuxCrystal, Suhas K, technusm1, Thomas Gerot, tyteen4a03, Wladimir Palant, zvone
42	Définition de fonctions avec des arguments de liste	zenlc2000
43	Déploiement	Gal Dreiman, Iancnorden, Wayne Werner
44	Dérogation de méthode	DeepSpace, James
45	Des classes	Aaron Hall, Ahsanul Haque, Akshat Mahajan, Andrzej Pronobis , Anthony Pham, Avantol13, Camsbury, cfi, Community, Conrad.Dean, Daksh Gupta, Darth Shadow, Dartmouth, depperm, Elazar, Ffisegydd, Haris, Igor Raush, InitializeSahib, J F, jkdev, jlarsch, John Militer, Jonas S, Jonathan, Kallz, KartikKannapur, Kevin Brown, Kinifwyne, Leo, Liteye, lmiguelvargasf, Mailerdaimon, Martijn Pieters, Massimiliano Kraus, Mattew Whitt, MrP01, Nathan Arthur, ojas mohril, Pasha, Peter Steele, pistache, Preston, pylang, Richard Fitzhugh, rohittk239, Rushy Panchal, Sempoo, Simplans, Soumendra Kumar Sahoo, SuperBiasedMan, techydesigner, then0rTh, Thomas Gerot, Tony Suffolk 66, tox123, UltraBob, user2314737, wrwrwr, Yogendra Sharma
46	Des exceptions	Adrian Antunez, Alessandro Trinca Tornidor, Alfe, Andy, Benjamin Hodgson, Brian Rodriguez, BusyAnt, Claudiu, driax, Elazar, flazzarini, ghostarbeiter, Ilia Barahovski, J F, Marco Pashkov, muddyfish, noɥʎdʎzɐɹƆ, Paul Weaver, Rahul Nair, RamenChef, Shawn Mehan, Shiven, Shkelqim Memolla, Simplans, Slickytail, Stephen Leppik, Sudip Bhandari, SuperBiasedMan, user2314737
47	Descripteur	bbayles, cizixs, Nemo, pylang, SuperBiasedMan
48	Déstructurer la liste (aka emballage et déballage)	J F, sth, zmo
49	dictionnaire	Amir Rachum, Anthony Pham, APerson, ArtOfCode, BoppreH, Burhan Khalid, Chris Mueller, cizixs, depperm, Ffisegydd,

		Gareth Latty, Guy, helpful, iBelieve, Igor Raush, Infinity, James , JGreenwell, jonrsharpe, Karsten 7., kdopen, machine yearning, Majid, mattgathu, Mechanic, MSeifert, muddyfish, Nathan, nlsdfnbch, noɥʎdʎzɐɔ, ronrest, Roy Iacob, Shawn Mehan, Simplans, SuperBiasedMan, TehTris, Valentin Lorentz, viveksyngh, Xavier Combelle
50	Différence entre module et package	DeepSpace, Simplans, tjohnson
51	Distribution	Alessandro Trinca Tornidor, JGreenwell, metahost, Pigman168 , RamenChef, Stephen Leppik
52	Django	code_geek, orvi
53	Données binaires	Eleftheria, evuez, mnoronha
54	Douilles	David Cullen, Dev, MattCorr, nlsdfnbch, Rob H, StuxCrystal, textshell, Thomas Gerot, Will
55	Échancrure	Alessandro Trinca Tornidor, depperm, J F, JGreenwell, Matt Giltaji, Pasha, RamenChef, Stephen Leppik
56	Écrire dans un fichier CSV à partir d'une chaîne ou d'une liste	Hriddhi Dey, Thomas Crowley
57	Écrire des extensions	Dartmouth, J F, mattgathu, Nathan Osman, techydesigner, ygram
58	Empiler	ADITYA, boboquack, Chromium, cjds, depperm, Hannes Karppila, JGreenwell, Jonatan, kdopen, OliPro007, orvi, SashaZd, Sнађошƒa, textshell, Thomas Ahle, user2314737
59	Enregistrement	Gal Dreiman, Jörn Hees, sxnwlfkk
60	Ensemble	Andrzej Pronobis, Andy Hayden, Bahrom, Cimbali, Cody Piersall, Conrad.Dean, Elazar, evuez, J F, James, Or East, pylang, RahulHP, RamenChef, Simplans, user2314737
61	Entrée et sortie de base	Doraemon, GoatsWearHats, J F, JNat, Marco Pashkov, Mark Miller, Martijn Pieters, Nathaniel Ford, Nicolás, pcurry, pzp, SashaZd, SuperBiasedMan, Vilmar
62	Enum	Andy, Elazar, evuez, Martijn Pieters, techydesigner
63	environnement virtuel avec virtualenvwrapper	Sirajus Salayhin

64	Environnement virtuel Python - virtualenv	Vikash Kumar Jain
65	Environnements virtuels	Adrian17, Artem Kolontay, ArtOfCode, Bhargav, brennan, Dair, Daniil Ryzhkov, Darkade, Darth Shadow, edwinksl, Fernando, ghostarbeiter, ha_1694, Hans Then, Iancnorden, J F, Majid, Marco Pashkov, Matt Giltaji, Mattew Whitt, nehemiah, Nuhil Mehdy, Ortomala Lokni, Preston, pylang, qwertyuip9, RamenChef, Régis B., Sebastian Schrader, Serenity, Shantanu Alshi, Shrey Gupta, Simon Fraser, Simplans, wrwrwr , ychaouche, zopieux, zvezda
66	étagère	Biswa_9937
67	Événements envoyés par le serveur Python	Nick Humrich
68	Exceptions du Commonwealth	Juan T, TemporalWolf
69	Exécution de code dynamique avec `exec` et` eval`	Antti Haapala, Ilja Everilä
70	Exponentiation	Anthony Pham, intboolstring, jtbandes, Luke Taylor, MSeifert, Pasha, supersam654
71	Expressions idiomatiques	Benjamin Hodgson, Elazar, Faiz Halde, J F, Lee Netherton, loading..., Mister Mister
72	Expressions régulières (Regex)	Aidan, alejosocorro, andandandand, Andy Hayden, ashes999, B8vrede, Claudiu, Darth Shadow, driax, Fermi paradox, ganesh gadila, goodmami, Jan, Jeffrey Lin, jonrsharpe, Julien Spronck, Kevin Brown, Md.Sifatul Islam, Michael M., mnoronha , Nander Speerstra, nrusch, Or East, orvi, regnarg, sarvajeetsuman, Simplans, SN Ravichandran KR, SuperBiasedMan, user2314737, zondo
73	fichier temporaire NamedTemporaryFile	Alessandro Trinca Tornidor, amblina, Kevin Brown, Stephen Leppik
74	Fichiers de décompression	andrew
75	Fichiers de données externes d'entrée, de sous-ensemble et de sortie à l'aide de	Mark Miller

	Pandas	
76	Fichiers et dossiers E / S	Ajean, Anthony Pham, avb, Benjamin Hodgson, Bharel, Charles, crhodes, David Cullen, Dov, Esteis, ilse2005, isvforall, jfsturtz, Justin, Kevin Brown, mattgathu, MSeifert, nlsdfnbch, Ozair Kafray, PYPL, pzp, RamenChef, Ronen Ness, rrao, Serenity, Simplans, SuperBiasedMan, Tasdik Rahman, Thomas Gerot, Umibozu, user2314737, Will, WombatPM, xgord
77	Filtre	APerson, cfi, J Atkin, MSeifert, rajah9, SuperBiasedMan
78	Fonction de la carte	APerson, cfi, Igor Raush, Jon Ericson, Karl Knechtel, Marco Pashkov, MSeifert, noɥʎdʎzɐƆ, Parousia, Simplans, SuperBiasedMan, tlama, user2314737
79	Fonctions partielles	FrankBr
80	Formatage de chaîne	4444, Aaron Christiansen, Adam_92, ADITYA, Akshit Soota, aldanor, alecxe, Alessandro Trinca Tornidor, Andy Hayden, Ani Menon, B8vrede, Bahrom, Bhargav, Charles, Chris, Darth Shadow, Dartmouth, Dave J, Delgan, dreftymac, evuez, Franck Dernoncourt, Gal Dreiman, gerrit, Giannis Spiliopoulos, GiantsLoveDeathMetal, goyalankit, Harrison, James Elderfield, Jean-Francois T., Jeffrey Lin, jetpack_guy, JL Peyret, joel3000, Jonatan, JRodDynamite, Justin, Kevin Brown, knight, krato, Marco Pashkov, Mark, Matt, Matt Giltaji, mu , MYGz, Nander Speerstra, Nathan Arthur, Nour Chawich, orion_tvv, ragesz, SashaZd, Serenity, serv-inc, Simplans, Slayther, Sometowngeek, SuperBiasedMan, Thomas Gerot, tobias_k, Tony Suffolk 66, UloPe, user2314737, user312016, Vin, zondo
81	Formatage de date	surfthecity
82	Générateurs	2Cubed, Ahsanul Haque, Akshat Mahajan, Andy Hayden, Arthur Dent, ArtOfCode, Augustin, Barry, Chankey Pathak, Claudiu, CodenameLambda, Community, deeenes, Delgan, Devesh Saini, Elazar, ericmarkmartin, Ernir, ForceBru, Igor Raush, Ilia Barahovski, J0HN, jackskis, Jim Fasarakis Hilliard, Juan T, Julius Bullinger, Karl Knechtel, Kevin Brown, Kronen, Luc M, Lyndsy Simon, machine yearning, Martijn Pieters, Matt Giltaji, max, MSeifert, nlsdfnbch, Pasha, Pedro, PsyKzz, pzp, satsumas, sevenforce, Signal, Simplans, Slayther, StuxCrystal, tversteeg, Valentin Lorentz, Will, William Merrill, xtreak, Zaid Ajaj, zarak, λuser
83	Gestionnaires de contexte (déclaration	Abhijeet Kasurde, Alessandro Trinca Tornidor, Andy Hayden, Antoine Bolvy, carrdelling, Conrad.Dean, Dartmouth, David

	«avec»)	Marx, DeepSpace, Elazar, Kevin Brown, magu_, Majid, Martijn Pieters, Matthew, nlsdfnbch, Pasha, Peter Brittain, petrs, Shuo, Simplans, SuperBiasedMan, The_Cthulhu_Kid, Thomas Gerot, tyteen4a03, user312016, Valentin Lorentz, vaultah, λuser
84	hashlib	Mark Omo, xiaoyi
85	ijson	Prem Narain
86	Implémentations non officielles de Python	Jacques de Hooge, Squidward
87	Importation de modules	angussidney, Anthony Pham, Antonis Kalou, Brett Cannon, BusyAnt, Casebash, Christian Ternus, Community, Conrad.Dean, Daniel, Dartmouth, Esteis, Ffisegydd, FMc, Gerard Roche, Gideon Buckwalter, J F, JGreenwell, Kinifwyne, languitar, Lex Scarisbrick, Matt Giltaji, MSeifert, niyasc, nlsdfnbch, Paulo Freitas, pylang, Rahul Nair, Saiful Azad, Serenity, Simplans, StardustGogeta, StuxCrystal, SuperBiasedMan, techydesigner, the_cat_lady, Thomas Gerot, Tony Meyer, Tushortz, user2683246, Valentin Lorentz, Valor Naram, vaultah, wnnmaw
88	Incompatibilités entre Python 2 et Python 3	671620616, Abhishek Kumar, Akshit Soota, Alex Gaynor, Allan Burleson, Alleo, Amarpreet Singh, Andy Hayden, Ani Menon, Antoine Bolvy, AntsySysHack, Antti Haapala, Antwan, arekolek , Ares, asmeurer, B8vrede, Bakuriu, Bharel, Bhargav Rao, bignose, bitchaser, Bluethon, Cache Staheli, Cameron Gagnon , Charles, Charlie H, Chris Sprague, Claudiu, Clayton Wahlstrom, cʟᴅsᴇᴇᴅ, Colin Yang, Cometsong, Community, Conrad.Dean, danidee, Daniel Stradowski, Darth Shadow, Dartmouth, Dave J, David Cullen, David Heyman, deeenes, DeepSpace, Delgan, DoHe, Duh-Wayne-101, Dunno, dwanderson, Ekeyme Mo, Elazar, enderland, enrico.bacis, erewok, ericdwang, ericmarkmartin, Ernir, ettanany, Everyone_Else, evuez, Franck Dernoncourt, Fred Barclay, garg10may, Gavin, geoffspear, ghostarbeiter, GoatsWearHats, H. Pauwelyn, Haohu Shen, holdenweb, iScrE4m, Iván C., J F, J. C. Leitão, James Elderfield, James Thiele, jarondl, jedwards, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, Jimmy Song, John Slegers, Jojodmo, jonrsharpe, Josh, Juan T, Justin, Justin M. Ucar, Kabie, kamalbanga, Karl Knechtel, Kevin Brown, King's jester, Kunal Marwaha, Lafexlos, lenz, linkdd, l'L'l, Mahdi, Martijn Pieters, Martin Thoma, masnun, Matt, Matt Dodge, Matt Rowland, Mattew Whitt, Max Feng, mgwilliams, Michael Recachinas, mkj, mnoronha, Moinuddin Quadri, muddyfish, Nathaniel Ford, niemmi, niyasc, noɥʎdʎzɐɹƆ, OrangeTux, Pasha, Paul Weaver, Paulo Freitas, pcurry,

		pktangyue, poppie, pylang, python273, Pythonista, RahulHP, Rakitić, RamenChef, Rauf, René G, rfkortekaas, rrao, Ryan, sblair, Scott Mermelstein, Selcuk, Serenity, Seth M. Larson, ShadowRanger, Simplans, Slayther, solarc, sricharan, Steven Hewitt, sth, SuperBiasedMan, Tadhg McDonald-Jensen, techydesigner, Thomas Gerot, Tim, tobias_k, Tyler, tyteen4a03, user2314737, user312016, Valentin Lorentz, Veedrac, Ven, Vinayak, Vlad Shcherbina, VPfB, WeizhongTu, Wieland, wim, Wolf, Wombatz, xtreak, zarak, zcb, zopieux, zurfyx, zvezda
89	Indexation et découpage	Alleo, amblina, Antoine Bolvy, Bonifacio2, Ffisegydd, Guy, Igor Raush, Jonatan, Martec, MSeifert, MUSR, pzp, RahulHP, Reut Sharabani, SashaZd, Sayed M Ahamad, SuperBiasedMan, theheadofabroom, user2314737, yurib
90	Interface de passerelle de serveur Web (WSGI)	David Heyman, Kevin Brown, Preston, techydesigner
91	Introduction à RabbitMQ en utilisant AMQPStorm	eandersson
92	Iterables et Iterators	4444, Conrad.Dean, demonplus, Ilia Barahovski, Pythonista
93	kivy - Framework Python multiplate-forme pour le développement NUI	dhimanta
94	l'audio	blueberryfields, Comrade SparklePony, frankyjuang, jmunsch, orvi, qwertyuip9, Stephen Leppik, Thomas Gerot
95	L'interpréteur (console de ligne de commande)	Aaron Christiansen, David, Elazar, Peter Shinners, ppperry
96	La déclaration de passage	Anaphory
97	La fonction d'impression	Beall619, Frustrated, Justin, Leon Z., lukewrites, SuperBiasedMan, Valentin Lorentz
98	La variable spéciale __name__	Annonymous, BusyAnt, Christian Ternus, jonrsharpe, Lutz Prechelt, Steven Elliott
99	Le débogage	Aldo, B8vrede, joel3000, Sardathrion, Sardorbek Imomaliev, Vlad Bezden

100	Le module base64	Thomas Gerot
101	Le module dis	muddyfish, user2314737
102	Le module local	Will, XonAether
103	Le module os	Andy, Christian Ternus, JelmerS, JL Peyret, mnoronha, Vinzee
104	Lecture et écriture CSV	Adam Matan, Franck Dernoncourt, Martin Valgur, mnoronha, ravigadila, Setu
105	Les fonctions	Adriano, Akshat Mahajan, AlexV, Andy, Andy Hayden, Anthony Pham, Arkady, B8vrede, Benjamin Hodgson, btel, CamelBackNotation, Camsbury, Chandan Purohit, ChaoticTwist, Charlie H, Chris Larson, Community, D. Alveno, danidee, DawnPaladin, Delgan, duan, duckman_1991, elegent , Elodin, Emma, EsmaeelE, Ffisegydd, Gal Dreiman, ghostarbeiter, Hurkyl, J F, James, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, jkitchen, Jossie Calderon, Justin, Kevin Brown, L3viathan, Lee Netherton, Martijn Pieters, Martin Thurau, Matt Giltaji, Mike - SMT, Mike Driscoll, MSeifert, muddyfish, Murphy4, nd., noɥʎdʎzɐɹƆ, Pasha, pylang, pzp, Rahul Nair, Severiano Jaramillo Quintanar, Simplans, Slayther, Steve Barnes, Steven Maude, SuperBiasedMan, textshell, then0rTh, Thomas Gerot, user2314737, user3333708, user405 , Utsav T, vaultah, Veedrac, Will, Will, zxxz, λuser
106	liste	Adriano, Alexander, Anthony Pham, Ares, Barry, blueenvelope, Bosoneando, BusyAnt, Çağatay Uslu, caped114, Chandan Purohit, ChaoticTwist, cizixs, Daniel Porteous, Darth Kotik, deeenes, Delgan, Elazar, Ellis, Emma, evuez, exhuma, Ffisegydd, Flickerlight, Gal Dreiman, ganesh gadila, ghostarbeiter, Igor Raush, intboolstring, J F, j3485, jalanb, James, James Elderfield, jani, jimsug, jkdev, JNat, jonrsharpe, KartikKannapur, Kevin Brown, Lafexlos, LDP, Leo Thumma, Luke Taylor, lukewrites, lxer, Majid, Mechanic, MrP01, MSeifert , muddyfish, n12312, noɥʎdʎzɐɹƆ, Oz Bar-Shalom, Pasha, Pavan Nath, poke, RamenChef, ravigadila, ronrest, Serenity, Severiano Jaramillo Quintanar, Shawn Mehan, Simplans, sirin, solarc, SuperBiasedMan, textshell, The_Cthulhu_Kid, user2314737, user6457549, Utsav T, Valentin Lorentz, vaultah , Will, wythagoras, Xavier Combelle
107	Liste de coupe (sélection de parties de listes)	Greg, JakeD
108	Liste des	3442, 4444, acdr, Ahsanul Haque, Akshay Anand, Akshit

	compréhensions	Soota, Alleo, Amir Rachum, André Laszlo, Andy Hayden, Ankit Kumar Singh, Antoine Bolvy, APerson, Ashwinee K Jha, B8vrede, bfontaine, Brian Cline, Brien, Casebash, Celeo, cfi, ChaoticTwist, Charles, Charlie H, Chong Tang, Community, Conrad.Dean, Dair, Daniel Stradowski, Darth Shadow, Dartmouth, David Heyman, Delgan, Dima Tisnek, eenblam, Elazar, Emma, enrico.bacis, EOL, ericdwang, ericmarkmartin, Esteis, Faiz Halde, Felk, Fermi paradox, Florian Bender, Franck Dernoncourt, Fred Barclay, freidrichen, G M, Gal Dreiman, garg10may, ghostarbeiter, GingerHead, griswolf, Hannele, Harry, Hurkyl, IanAuld, iankit, Infinity, intboolstring, J F, J0HN, James, JamesS, Jamie Rees, jedwards, Jeff Langemeier, JGreenwell, JHS, jjwatt, JKillian, JNat, joel3000, John Slegers, Jon, jonrsharpe, Josh Caswell, JRodDynamite, Julian, justhalf, Kamyar Ghasemlou, kdopen, Kevin Brown, KIDJourney, Kwarrtz, Lafexlos, lapis, Lee Netherton, Liteye, Locane, Lyndsy Simon, machine yearning, Mahdi, Marc, Markus Meskanen, Martijn Pieters, Matt, Matt Giltaji, Matt S, Mattew Whitt, Maximillian Laumeister, mbrig, Mirec Miskuf, Mitch Talmadge, Morgan Thrapp, MSeifert, muddyfish, n8henrie, Nathan Arthur, nehemiah, noɥʎʞʎzɐɹƆ, Or East, Ortomala Lokni, pabouk, Panda, Pasha, pktangyue, Preston, Pro Q, pylang, R Nar, Rahul Nair, rap-2-h, Riccardo Petraglia, rll, Rob Fagen, rrao, Ryan Hilbert, Ryan Smith, ryanyuyu, Samuel McKay, sarvajeetsuman, Sayakiss, Sebastian Kreft, Shoe, SHOWMEWHATYOUGOT, Simplans, Slayther, Slickytail, solidcell, StuxCrystal, sudo bangbang, Sunny Patel, SuperBiasedMan, syb0rg, Symmitchry, The_Curry_Man, theheadofabroom, Thomas Gerot, Tim McNamara, Tom Barron, user2314737, user2357112, Utsav T, Valentin Lorentz, Veedrac, viveksyngh, vog, W.P. McNeill, Will, Will, Wladimir Palant, Wolf, XCoder Real, yurib, Yury Fedorov, Zags, Zaz
109	Listes liées	Nemo
110	Manipulation de XML	4444, Brad Larson, Chinmay Hegde, Francisco Guimaraes, greuze, heyhey2k, Rob Murray
111	Mathématiques complexes	Adeel Ansari, Bosoneando, bpachev
112	Métaclasses	2Cubed, Amir Rachum, Antoine Pinsard, Camsbury, Community, driax, Igor Raush, InitializeSahib, Marco Pashkov, Martijn Pieters, Mattew Whitt, OozeMeister, Pasha, Paulo Scardine, RamenChef, Rob Bednark, Simplans, sisanared, zvone
113	Méthodes de chaîne	Amitay Stern, Andy Hayden, Ares, Bhargav Rao, Brien,

		BusyAnt, Cache Staheli, caped114, ChaoticTwist, Charles, Dartmouth, David Heyman, depperm, Doug Henderson, Elazar , ganesh gadila, ghostarbeiter, GoatsWearHats, idjaw, Igor Raush, Ilia Barahovski, j__, Jim Fasarakis Hilliard, JL Peyret, Kevin Brown, krato, MarkyPython, Metasomatism, Mikail Land, MSeifert, mu , Nathaniel Ford, OliPro007, orvi, pzp, ronrest, Shrey Gupta, Simplans, SuperBiasedMan, theheadofabroom, user1349663, user2314737, Veedrac, WeizhongTu, wnnmaw
114	Méthodes définies par l'utilisateur	Alessandro Trinca Tornidor, Beall619, mnoronha, RamenChef, Stephen Leppik, Sun Qingyao
115	Mixins	Doc, Rahul Nair, SashaZd
116	Modèles de conception	Charul, denvaar, djaszczurowski
117	Modèles en python	4444, Alessandro Trinca Tornidor, Fred Barclay, RamenChef, Ricardo, Stephen Leppik
118	Module aléatoire	Alex Gaynor, Andrzej Pronobis, Anthony Pham, Community, David Robinson, Delgan, giucal, Jim Fasarakis Hilliard, michaelrbock, MSeifert, Nobilis, ppperry, RamenChef, Simplans, SuperBiasedMan
119	Module Asyncio	2Cubed, Alessandro Trinca Tornidor, Cimbali, hiro protagonist, obust, pylang, RamenChef, Seth M. Larson, Simplans, Stephen Leppik, Udi
120	Module Collections	asmeurer, Community, Elazar, jmunsch, kon psych, Marco Pashkov, MSeifert, RamenChef, Shawn Mehan, Simplans, Steven Maude, Symmitchry, void, XCoder Real
121	Module de file d'attente	Prem Narain
122	Module Deque	Anthony Pham, BusyAnt, matsjoyce, ravigadila, Simplans, Thomas Ahle, user2314737
123	Module Functools	Alessandro Trinca Tornidor, enrico.bacis, flamenco, RamenChef, Shrey Gupta, Simplans, Stephen Leppik, StuxCrystal
124	Module Itertools	ADITYA, Alessandro Trinca Tornidor, Andy Hayden, balki, bpachev, Ffisegydd, jackskis, Julien Spronck, Kevin Brown, machine yearning, nlsdfnbch, pylang, RahulHP, RamenChef, Simplans, Stephen Leppik, Symmitchry, Wickramaranga, wnnmaw

125	Module JSON	Indradhanush Gupta, Leo, Martijn Pieters, pzp, theheadofabroom, Underyx, Wolfgang
126	Module Math	Anthony Pham, ArtOfCode, asmeurer, Christofer Ohlsson, Ellis , fredley, ghostarbeiter, Igor Raush, intboolstring, J F, James Elderfield, JGreenwell, MSeifert, niyasc, RahulHP, rajah9, Simplans, StardustGogeta, SuperBiasedMan, yurib
127	Module opérateur	MSeifert
128	module pyautogui	Damien, Rednivrug
129	Module Sqlite3	Chinmay Hegde, Simplans
130	Module Webbrowser	Thomas Gerot
131	Multithreading	Alu, cLDSEED, juggernaut, Kevin Brown, Kristof, mattgathu, Nabeel Ahmed, nlsdfnbch, Rahul, Rahul Nair, Riccardo Petraglia, Thomas Gerot, Will, Yogendra Sharma
132	Multitraitement	Alon Alexander, Nander Speerstra, unutbu, Vinzee, Will
133	Mutable vs immuable (et lavable) en Python	Cilyan
134	Neo4j et Cypher utilisant Py2Neo	Wingston Sharon
135	Noeud Liste liée	orvi
136	Objets de propriété	Alessandro Trinca Tornidor, Darth Shadow, DhiaTN, J F, Jacques de Hooge, Leo, Martijn Pieters, mnoronha, Priya, RamenChef, Stephen Leppik
137	Opérateurs booléens	boboquack, Brett Cannon, Dair, Ffisegydd, John Zwinck, Severiano Jaramillo Quintanar, Steven Maude
138	Opérateurs mathématiques simples	amin, blueenvelope, Bryce Frank, Camsbury, David, DeepSpace, Elazar, J F, James, JGreenwell, Jon Ericson, Kevin Brown, Lafexlos, matsjoyce, Mechanic, Milo P, MSeifert, numbermaniac, sarvajeetsuman, Simplans, techydesigner, Tony Suffolk 66, Undo, user2314737, wythagoras, Zenadix
139	Opérateurs sur les bits	Abhishek Jain, boboquack, Charles, Gal Dreiman, intboolstring , JakeD, JNat, Kevin Brown, Matías Brignone, nemesisfixx, poke, R Colmenares, Shawn Mehan, Simplans, Thomas Gerot, tmr232, Tony Suffolk 66, viveksyngh
140	Optimisation des	A. Ciclet, RamenChef, user2314737

	performances	
141	Oreiller	Razik
142	os.path	Claudiu, Fábio Perez, girish946, Jmills, Szabolcs Dombi, VJ Magar
143	Outil 2to3	Alessandro Trinca Tornidor, Dartmouth, Firix, Kevin Brown, Naga2Raja, Stephen Leppik
144	outil graphique	xiaoyi
145	Pandas Transform: préforme les opérations sur les groupes et concatène les résultats	Dee
146	par groupe()	Parousia, Thomas Gerot
147	Persistance python	RamenChef, user2728397
148	Pièges courants	abukaj, ADITYA, Alec, Alessandro Trinca Tornidor, Alex, Antoine Bolvy, Baaing Cow, Bhargav Rao, Billy, bixel, Charles, Cheney, Christophe Roussy, Dartmouth, DeepSpace, DhiaTN, Dilettant, fox, Fred Barclay, Gerard Roche, greatwolf, hiro protagonist, Jeffrey Lin, JGreenwell, Jim Fasarakis Hilliard, Lafexlos, maazza, Malt, Mark, matsjoyce, Matt Dodge, MervS, MSeifert, ncmathsadist, omgimanerd, Patrick Haugh, pylang, RamenChef, Reut Sharabani, Rob Bednark, rrao, SashaZd, Shihab Shahriar, Simplans, SuperBiasedMan, Tim D, Tom Dunbavan, tyteen4a03, user2314737, Will Vousden, Wombatz
149	pip: PyPI Package Manager	Andy, Arpit Solanki, Community, InitializeSahib, JNat, Mahdi, Majid, Matt Giltaji, Nathaniel Ford, Rápli András, SerialDev, Simplans, Steve Barnes, StuxCrystal, tlo
150	Polymorphisme	Benedict Bunting, DeepSpace, depperm, Simplans, skrrgwasme, Vinzee
151	Portée variable et liaison	Anthony Pham, davidism, Elazar, Esteis, Mike Driscoll, SuperBiasedMan, user2314737, zvone
152	PostgreSQL	Alessandro Trinca Tornidor, RamenChef, Stephen Leppik, user2027202827
153	Priorité de l'opérateur	HoverHell, JGreenwell, MathSquared, SashaZd, Shreyash S Sarnayak
154	Processus et threads	Claudiu, Thomas Gerot

155	Profilage	J F, keiv.fly, SashaZd
156	Programmation fonctionnelle en Python	Imran Bughio, mvis89, Rednivrug
157	Programmation IoT avec Python et Raspberry PI	dhimanta
158	py.test	Andy, Claudiu, Ffisegydd, Kinifwyne, Matt Giltaji
159	pyaudio	Biswa_9937
160	pygame	Anthony Pham, Aryaman Arora, Pavan Nath
161	Pyglet	Comrade SparklePony, Stephen Leppik
162	PyInstaller - Distribuer du code Python	ChaoticTwist, Eric, mnoronha
163	Python et Excel	bee-sting, Chinmay Hegde, GiantsLoveDeathMetal, hackvan, Majid, talhasch, user2314737, Will
164	Python Lex-Yacc	CLDSEED
165	Python Requests Post	Ken Y-N, RandomHash
166	Recherche	Dan Sanderson, Igor Raush, MSeifert
167	Reconnaissance optique de caractères	rassar
168	Récursivité	Bastian, japborst, JGreenwell, Jossie Calderon, mbomb007, SashaZd, Tyler Crompton
169	Réduire	APerson, Igor Raush, Martijn Pieters, MSeifert
170	Représentations de chaîne des instances de classe: méthodes __str__ et __repr__	Alessandro Trinca Tornidor, jedwards, JelmerS, RamenChef, Stephen Leppik
171	Réseau Python	atayenel, ChaoticTwist, David, Geeklhem, mattgathu, mnoronha, thsecmaniac
172	Sécurité et cryptographie	adeora, ArtOfCode, BSL-5, Kevin Brown, matsjoyce, SuperBiasedMan, Thomas Gerot, Wladimir Palant, wrwrwr

173	Sérialisation des données	Devesh Saini, Infinity, rfkortekaas
174	Sérialisation des données de pickle	J F, Majid, Or East, RahulHP, rfkortekaas, zvone
175	Serveur HTTP Python	Arpit Solanki, J F, jmunsch, Justin Chadwell, Mark, MervS, orvi , quantummind, Raghav, RamenChef, Sachin Kalkur, Simplans , techydesigner
176	setup.py	Adam Brenecki, amblina, JNat, ravigadila, strpeter, user2027202827, Y0da
177	Similitudes dans la syntaxe, différences de sens: Python vs JavaScript	user2683246
178	Sockets et cryptage / décryptage de messages entre le client et le serveur	Mohammad Julfikar
179	Sous-commandes CLI avec sortie d'aide précise	Alessandro Trinca Tornidor, anatoly techtonik, Darth Shadow
180	Surcharge	Andy Hayden, Darth Shadow, ericmarkmartin, Ffisegydd, Igor Raush, Jonas S, jonrsharpe, L3viathan, Majid, RamenChef, Simplans, Valentin Lorentz
181	sys	blubberdiblub
182	Tableaux	Andy, Pavan Nath, RamenChef, Vin
183	Tableaux multidimensionnels	boboquack, Buzz, rrao
184	Tas	ettanany
185	Test d'unité	Alireza Savand, Ami Tavory, antimatter15, Arpit Solanki, bijancn, Claudiu, Dartmouth, engineercoding, Ffisegydd, J F, JGreenwell, jmunsch, joel3000, Kevin Brown, Kinifwyne, Mario Corchero, Matt Giltaji, Mattew Whitt, mgilson, muddyfish, pylang, strpeter
186	tkinter	Dartmouth, rlee827, Thomas Gerot, TidB
187	Tortue Graphiques	Luca Van Oort, Stephen Leppik

188	Tracer avec Matplotlib	Arun, user2314737
189	Travailler autour du verrou d'interprète global (GIL)	Scott Mermelstein
190	Travailler avec des archives ZIP	Chinmay Hegde, ghostarbeiter, Jeffrey Lin, SuperBiasedMan
191	Tri, minimum et maximum	Antti Haapala, APerson, GoatsWearHats, Mirec Miskuf, MSeifert, RamenChef, Simplans, Valentin Lorentz
192	Tuple	Anthony Pham, Antoine Bolvy, BusyAnt, Community, Elazar, James, Jim Fasarakis Hilliard, Joab Mendes, Majid, Md.Sifatul Islam, Mechanic, mezzode, nlsdfnbch, noɥʎdʎzɐɔ, Selcuk, Simplans, textshell, tobias_k, Tony Suffolk 66, user2314737
193	Type conseils	alecxe, Annonymous, Antti Haapala, Elazar, Jim Fasarakis Hilliard, Jonatan, RamenChef, Seth M. Larson, Simplans, Stephen Leppik
194	Types de données immuables (int, float, str, tuple et frozensets)	Alessandro Trinca Tornidor, FazeL, Ganesh K, RamenChef, Stephen Leppik
195	Types de données Python	Gavin, lorenzofeliz, Pike D., Rednivrug
196	Unicode	wim
197	Unicode et octets	Claudiu, KeyWeeUsr
198	urllib	Amitay Stern, ravigadila, sth, Will
199	Utilisation du module "pip": PyPI Package Manager	Zydnar
200	Utiliser des boucles dans les fonctions	naren
201	Vérification de l'existence du chemin et des autorisations	Esteis, Marlon Abeykoon, mnoronha, PYPL
202	Visualisation de données avec Python	Aquib Javed Khan, Arun, ChaoticTwist, cledoux, Ffisegydd, ifma

203	Vitesse du programme Python	ADITYA, Antonio, Elodin, Neil A., Vinzee
204	Web grattant avec Python	alecxe, Amitay Stern, jmunsch, mrtuovinen, Ni., RamenChef, Saiful Azad, Saqib Shamsi, Simplans, Steven Maude, sth, sytech, talhasch, Thomas Gerot
205	Websockets	2Cubed, Stephen Leppik, Tyler Gubala