



無料電子ブック

学習

Python Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#python

.....	1
1: Python	2
.....	2
.....	3
Python 3.x.....	3
Python 2.x.....	3
Examples.....	3
.....	3
Python.....	4
IDLEPython	4
Hello World Python.....	5
Python	6
.....	7
.....	7
.....	7
.....	8
.....	12
IDLE - Python GUI.....	13
.....	14
.....	14
.....	14
.....	15
.....	15
.....	15
.....	15
.....	16
.....	17
.....	17
.....	18
.....	18
.....	18
.....	

.....	23
.....	24
.....	25
.....	29
.....	31
- strrepr.....	32
repr.....	32
str.....	32
pip.....	33
/.....	33
.....	34
.....	34
Python 2.7.x3.x.....	34
2: "pip"PyPI Package Manager.....	38
.....	38
.....	38
Examples.....	38
.....	39
ImportError.....	39
.....	40
3: * args** kwargs.....	41
.....	41
h11.....	41
h12.....	41
h13.....	41
Examples.....	42
* args.....	42
** kwargs.....	42
* args.....	43
** kwargs.....	43

* args.....	44
.....	44
kwarg.....	45
** kwargs.....	45
4: __name__.....	46
.....	46
.....	46
Examples.....	46
__name__ == '__main__'.....	46
1.....	46
2.....	46
function_class_or_module .__ name__.....	47
.....	48
5: `exec` `eval`.....	49
.....	49
.....	49
.....	49
Examples.....	49
exec.....	49
eval.....	50
.....	50
eval.....	50
ast.literal_evalPython.....	50
execevalast.literal_eval.....	51
6: 2to3.....	52
.....	52
.....	52
.....	53
Examples.....	53
.....	53
Unix.....	53
Windows.....	53

Unix.....	54
Windows.....	54
7: AMQPStormRabbitMQ.....	55
.....	55
Examples.....	55
RabbitMQ.....	55
RabbitMQ.....	56
RabbitMQ.....	56
8: ArcPy.....	59
.....	59
Examples.....	59
1.....	59
createDissolvedGDBGdb.....	59
9: Asyncio.....	60
Examples.....	60
.....	60
.....	61
UVLoop.....	62
.....	62
.....	62
.....	62
WebSocket.....	63
asyncio.....	64
10: base64.....	65
.....	65
.....	65
.....	65
.....	67
Examples.....	67
Base64.....	67
Base32.....	69

Base16.....	69
ASCII85.....	70
Base85.....	70
11: ChemPy - python.....	72
.....	72
Examples.....	72
.....	72
.....	72
.....	72
.....	73
.....	73
.....	73
12: configparser.....	75
.....	75
.....	75
.....	75
Examples.....	75
.....	75
.....	75
13: CSV.....	77
Examples.....	77
TSV.....	77
Python.....	77
.....	77
.....	77
14: ctypes.....	78
.....	78
Examples.....	78
.....	78
.....	78
.....	78

.....	79
ctypes.....	79
ctypes.....	80
ctypes.....	80
.....	81
15: Deque.....	83
.....	83
.....	83
.....	83
Examples.....	83
deque.....	83
.....	83
.....	84
.....	85
16: Functools.....	86
Examples.....	86
.....	86
total_ordering.....	86
.....	87
lru_cache.....	87
cmp_to_key.....	88
17: groupby.....	89
.....	89
.....	89
.....	89
.....	89
Examples.....	89
1.....	89
2.....	90
3.....	91
4.....	92
18: GZip.....	93

.....	93
Examples.....	93
GNU zip.....	93
19: HTML.....	94
Examples.....	94
BeautifulSoup.....	94
BeautifulSoupCSS.....	94
PyQuery.....	95
20: ijson.....	96
.....	96
Examples.....	96
.....	96
21: Itertools.....	97
.....	97
Examples.....	97
.....	97
.....	98
itertools.product.....	98
itertools.count.....	99
itertools.takewhile.....	100
itertools.dropwhile.....	101
2.....	101
itertools.....	102
.....	102
itertools.repeat.....	103
.....	103
.....	103
itertools.permutations.....	104
22: JSON.....	105
.....	105
.....	105
.....	105

.....	105
.....	105
.....	106
.....	106
.....	106
.....	106
Examples.....	107
Python dictJSON.....	107
JSONPython dict.....	107
.....	107
.....	108
`load`` dump` `dump`` dump '.....	108
JSON `json.tool`.....	109
JSON.....	110
.....	110
.....	110
.....	110
JSON.....	111
23: kivy - NUIPython.....	112
.....	112
Examples.....	112
.....	112
24: Matplotlib.....	115
.....	115
Examples.....	115
Matplotlib.....	115
.....	116
MATLAB.....	117
.....	118
XYtwinx.....	119
twinyYX.....	121
25: MutablePythonHashable.....	124

Examples.....	124
.....	124
.....	124
.....	125
.....	125
.....	126
.....	126
.....	127
26: os.path.....	128
.....	128
.....	128
Examples.....	128
.....	128
.....	128
.....	129
.....	129
.....	129
.....	129
27: os.....	131
.....	131
.....	131
.....	131
Examples.....	131
.....	131
.....	131
.....	131
.....	132
POSIX.....	132
.....	132
makedirs -	132
28: Pandas Transform.....	134
Examples.....	134

.....	134
.....	134
transform.....	134
.....	135
transformtransform.....	135
29: Pandas.....	137
.....	137
Examples.....	137
Pandas.....	137
30: pipPyPI.....	139
.....	139
.....	139
.....	139
Examples.....	140
.....	140
.....	140
.....	140
`pip`.....	140
.....	141
Linux.....	141
Windows.....	141
requirements.txt.....	142
virtualenvrequirements.txt.....	142
pipPython.....	142
.....	143
.....	144
.....	144
31: PostgreSQL.....	147
Examples.....	147
.....	147
pip.....	147
.....	

32: py.test.....148

 Examples.....148

 py.test.....148

 148

 148

 148

 149

 149

 py.test.....150

 152

33: Py2NeoNeo4jCypher.....154

 Examples.....154

 154

 Neo4j.....154

 Neo4j.....154

 1.....155

 2.....155

 Cypher.....155

34: pyautogui.....157

 157

 Examples.....157

 157

 157

 157

35: PyInstaller - Python.....158

 158

 158

 Examples.....158

 158

 Pyinstaller.....159

 1.....159

.....	159
.....	159
.....	159
36: Python 2Python 3	161
.....	161
.....	161
Examples.....	162
.....	162
Unicode.....	163
.....	165
Reduce.....	167
xrange.....	167
.....	169
.....	169
.....	171
.next.....	173
.....	173
.....	174
.....	175
execPython 3.....	176
Python 2hasattr.....	176
.....	177
.....	178
.....	178
Python 3.....	178
<> `=repr.....	179
16/.....	180
Python 3cmp.....	180
.....	181
.....	182
filtermapzip.....	183
/.....	183
.....	

184	
I / O.....	185
round.....	186
.....	186
round.....	186
.....	187
.....	187
int.....	188
.....	188
37: Python HTTP Server.....	190
Examples.....	190
HTTP.....	190
.....	190
SimpleHTTPServerAPI.....	192
BaseHTTPRequestHandlerGETPOSTPUT.....	193
38: Python Lex-Yacc.....	195
.....	195
.....	195
Examples.....	195
PLY.....	195
PLY -	195
1Lex.....	197
.....	198
h21.....	199
h22.....	199
h23.....	199
h24.....	199
h25.....	200
h26.....	200
h27.....	200
h28.....	200
h29.....	200

h210.....	201
2Yacc.....	201
.....	202
h211.....	203
39: Python Server Sent.....	204
.....	204
Examples.....	204
SSE.....	204
Asyncio SSE.....	204
40: Pythonpyserial.....	205
.....	205
.....	205
.....	205
Examples.....	205
.....	205
.....	205
.....	206
41: Python.....	207
.....	207
Examples.....	207
.....	207
.....	207
.....	207
.....	207
.....	208
.....	208
42: Python.....	209
.....	209
Examples.....	209
ssh.....	209
43: Python.....	210

.....	210
Examples.....	210
.....	210
wrapper.....	210
44: Python.....	212
.....	212
Examples.....	212
.....	212
.....	212
.....	212
.....	212
45: PythonExcel.....	214
Examples.....	214
Excel.....	214
OpenPyXL.....	214
xlswriterExcel.....	215
xlrdExcel.....	217
Excelxlswriter.....	218
46: PythonRaspberry PiIoT.....	220
Examples.....	220
-	220
47: PythonSQL Server.....	223
Examples.....	223
.....	223
48: PythonWeb.....	225
.....	225
.....	225
WebPython.....	225
.....	225
requests.....	225
requests-cache.....	225
scrapy.....	225

selenium.....	225
HTML.....	225
BeautifulSoup.....	225
lxml.....	226
Examples.....	226
lxml.....	226
Web.....	226
Scrapy.....	226
Scrapy.....	227
BeautifulSoup4.....	228
Selenium WebDriver.....	228
urllib.requestWeb.....	228
.....	229
49: Python.....	230
Examples.....	230
Matplotlib.....	230
.....	231
MayaVI.....	234
.....	235
50: Python.....	238
.....	238
Examples.....	238
Python/.....	238
HTTP.....	238
TCP.....	239
UDP.....	240
HttpServer.....	240
51: Python.....	242
Examples.....	242
except.....	242
.....	242
.....	243

52: Python	245
Examples	245
.....	245
.....	245
.....	245
.....	245
.....	246
.....	246
53: Python	248
Examples	248
.....	248
.....	248
54: Python	249
.....	249
Examples	249
.....	249
.....	249
.....	250
55: Python	252
.....	252
.....	252
Examples	252
Python	252
.....	253
56: Python	254
.....	254
Examples	254
.....	254
PyPI	255
.pyirc	255
testpyi	255

.....	256
PyPI	256
.....	256
Readme.....	257
.....	257
.....	257
57: PythonPost	258
.....	258
Examples.....	258
.....	258
.....	259
.....	260
.....	260
.....	261
.....	262
58: PythonC	263
.....	263
.....	263
Examples.....	264
CPython.....	264
PythonC.....	264
59: PythonWindows	267
.....	267
Examples.....	267
Python.....	267
Flask Web.....	268
60: Python - virtualenv	270
.....	270
Examples.....	270
.....	270
.....	270

Virtualenv	270
virtualenv	271
61: setup.py	272
.....	272
.....	272
Examples	272
setup.py	272
Python	273
setup.py	274
.....	274
62: Sqlite3	276
Examples	276
Sqlite3 -	276
.....	276
63: sys	278
.....	278
.....	278
.....	278
Examples	278
.....	278
.....	278
.....	278
.....	279
64: tempfile NamedTemporaryFile	280
.....	280
Examples	280
.....	280
65: Unicode	281
Examples	281
.....	281
66: Unicode	282
.....	282

.....	282
Examples.....	282
.....	282
Unicode.....	282
.....	283
/.....	283
.....	283
.....	284
.....	284
I/O.....	284
67: virtualenvwrapper.....	285
.....	285
Examples.....	285
virtualenvwrapper.....	285
68: Websockets.....	287
Examples.....	287
aiohttp.....	287
aiohttp.....	287
Websocket.....	288
69: WebWSGI.....	290
.....	290
Examples.....	290
.....	290
70: Web.....	292
.....	292
.....	292
.....	292
.....	293
Examples.....	293
URL.....	293
URL.....	294

71: Windowsvirtualenvwrapper	296
Examples	296
virtualenvwrapper	296
72: XML	298
	298
Examples	298
ElementTree	298
XML	298
XML	299
iterparseXML	299
XPathXML	300
73: ZIP	302
	302
	302
Examples	302
Zip	302
Zipfile	302
zip	303
	303
74:	305
Examples	305
	305
	306
iterable	306
iterable1	307
1	307
	307
75:	308
Examples	308
	308
	308
Python	309

PYTHONSTARTUP	309
.....	309
.....	310
76:	312
.....	312
.....	312
.....	312
Examples	312
.....	312
.....	313
.....	314
__getitem__ __setitem__ __delitem__	314
.....	315
.....	316
.....	316
77:	318
Examples	318
.....	318
.....	318
.....	319
.....	319
78:	321
Examples	321
HTTP GET	321
Python 2	321
Python 3	321
HTTP POST	322
Python 2	322
Python 3	322
.....	322
79:	324
Examples	324

Pyglet.....	324
WAV.....	324
winsound.....	324
.....	324
pythonffmpeg.....	324
Windows.....	325
80:	326
Examples.....	326
/.....	326
.....	327
.....	328
.....	328
.....	329
81:	332
.....	332
.....	332
Examples.....	333
.....	333
82:	334
Examples.....	334
iterablecollections.Counter.....	334
-scollections.Counter.most_common.....	334
1list.counttuple.count.....	335
str.count.....	335
numpy.....	335
83: /	337
.....	337
Examples.....	337
.....	337
.....	337
84:	339
.....	339

.....	339
Examples.....	341
.....	341
.....	342
del.....	343
.....	343
.....	344
.....	344
.....	344
.....	345
85:	347
.....	347
Examples.....	347
.....	347
86: /.....	348
.....	348
.....	348
Examples.....	351
.....	351
.....	353
87:	356
.....	356
Examples.....	356
.....	356
.....	357
.....	357
.....	358
.....	361
.....	362
.....	363
.....	365
.....	365

.....	367
.....	368
.....	368
.....	369
.....	371
.....	373
88: __str__ repr__	375
.....	375
.....	375
.....	375
Examples.....	375
.....	375
.....	377
1	377
2	378
.....	379
.....	380
eval-round-trip__repr__.....	380
89:	382
.....	382
Examples.....	382
PyDotPlus.....	382
.....	382
PyGraphviz.....	383
90: GIL	385
.....	385
GIL	385
GIL	385
GIL	385
GIL	386
.....	

Examples.....	386
.....	386
BeazleyGIL.....	386
Cython nogil.....	387
BeazleyGIL.....	387
.....	388
91:	389
.....	389
Examples.....	389
.....	389
92:	390
.....	390
Examples.....	390
argparseHello world.....	390
docopt.....	391
argparse.....	391
argv.....	392
argparse.....	393
argparse.add_argument_group.....	393
docoptdocopt_dispatch.....	394
93:	396
.....	396
.....	396
Examples.....	396
.....	396
docstring.....	396
.....	397
.....	397
docstrings.....	397
docstring.....	398
.....	

PEP 257.....	398
.....	399
Google Python.....	399
94:	401
.....	401
.....	401
Examples.....	401
.....	401
collections.defaultdict.....	403
.OrderedDict.....	404
collections.namedtuple.....	405
collections.deque.....	405
.ChainMap.....	407
95: "with"	409
.....	409
.....	409
.....	409
Examples.....	409
with.....	410
.....	410
.....	410
.....	411
.....	412
.....	413
96:	414
.....	414
.....	414
Examples.....	414
.....	414
Popen.....	414
.....	414

.....	415
.....	415
.....	415
.....	415
.....	416
.....	416
97:	417
.....	417
Examples.....	417
DjangoHello World.....	417
98:	419
.....	419
.....	419
.....	419
Examples.....	419
ListStack.....	419
.....	420
99:	422
.....	422
.....	422
.....	422
Examples.....	422
.....	422
.....	422
.....	423
.....	423
pycrypto.....	424
pycryptoRSA.....	425
pycryptoRSA.....	426
100:	427
.....	427
.....	

Examples.....427

.....427

.....428

.....429

.....429

.....430

.....430

.....430

.....430

.....430

.....431

.....431

.....431

.....432

101: 433

.....433

.....433

Examples.....433

 UDP.....433

 UDP.....433

 TCP.....434

 TCP.....435

 LinuxRaw.....436

102: 438

.....438

Examples.....438

 if / else.....438

.....438

.....439

.....440

103:	442
Examples	442
.....	442
104:	443
.....	443
.....	443
Examples	443
.....	443
.....	443
.....	444
.....	445
NamedTuple	445
.....	446
105:	447
.....	447
.....	447
.....	447
Examples	447
.....	447
.....	447
.....	448
.....	449
.....	449
.....	451
.....	451
.....	451
.....	451
.....	452
.....	452
.....	452
.....	452

106:	453
Examples	453
.....	453
.....	454
107:	455
Examples	455
dis	455
Python	455
.....	456
108:	457
Examples	457
.....	457
.....	457
.....	457
.....	457
.....	458
109:	459
.....	459
.....	459
.....	459
Examples	459
JSON	460
Pickle	460
110:	462
.....	462
Examples	462
MySQLdbMySQL	462
SQLite	463
SQLite	464
.....	464
h212	464
Connection	464

Cursor.....	465
SQLitePython.....	468
psycopg2PostgreSQL.....	469
.....	469
.....	469
.....	469
Oracle.....	470
.....	472
sqlalchemy.....	472
111:	474
.....	474
.....	474
.....	474
Examples.....	474
.....	474
.....	475
.....	476
.....	476
.....	477
.....	477
.....	477
.....	478
.....	478
.....	478
.....	479
.....	479
.....	480
112:	481
.....	481
Examples.....	481
.....	481
.....	

.....	484
113:	487
Examples.....	487
Python_pdb_.....	487
IPythonipdb.....	488
.....	489
114:	490
.....	490
.....	490
Examples.....	490
tkinter.....	490
.....	491
.....	491
.....	492
.....	492
115:	494
.....	494
.....	494
Examples.....	494
I / O.....	494
I / O.....	495
116:	497
.....	497
.....	497
.....	497
Examples.....	497
pygame.....	497
Pygame.....	498
.....	498
.....	498
.....	

117:	500
.....	500
Examples.....	500
.....	500
.....	500
.....	500
118:	502
.....	502
.....	502
Examples.....	503
.....	503
.....	504
119:	505
.....	505
Examples.....	505
os.access.....	505
120:	506
.....	506
Examples.....	506
MD5.....	506
OpenSSL.....	507
121:	508
.....	508
Examples.....	508
.....	508
122:	510
Examples.....	510
.....	510
.....	510
123:	512
.....	

.....	512
.....	512
Pickleable	512
pickle	512
Examples.....	513
Pickle.....	513
.....	513
.....	513
.....	513
.....	514
124:	516
.....	516
Examples.....	516
Hello World in Pyglet.....	516
Pyglet.....	516
Pyglet.....	516
OpenGLPyglet.....	516
PygletOpenGL.....	517
125:	518
.....	518
.....	518
Examples.....	518
AND.....	518
OR.....	518
.....	519
.....	519
.....	520
NOT.....	520
.....	522
126: I / O	523
.....	

.....	523
.....	523
.....	523
.....	523
Examples	524
.....	524
1.....	526
.....	526
.....	527
.....	528
.....	528
.....	529
.....	529
.....	530
mmap.....	530
.....	530
.....	531
127:	532
.....	532
Examples	532
Python ZipFile.extractallZIP	532
Python TarFile.extractalltarball	532
128:	533
.....	533
.....	533
.....	533
Examples	533
.....	533
.....	534
.....	534
filterfalseifilterfalse	535

129:	536
Examples	536
.....	536
.....	536
.....	537
.....	537
`and` or`	538
.....	538
130:	539
Examples	539
.....	539
.....	540
131:	542
.....	542
.....	542
Examples	542
.....	542
URL	543
HTTP	543
.....	544
Jinja	545
.....	546
URL	546
.....	546
.....	547
132: Python	548
Examples	548
.....	548
.....	548
.....	549
.....	549
.....	550

133:	552
.....	552
Examples.....	552
.....	552
.....	554
.....	554
.....	554
.....	555
134:	557
.....	557
Examples.....	557
@property.....	557
@property.....	557
gettersetterdeleter.....	558
.....	558
135:	561
Examples.....	561
IPython%% timeit.....	561
timeit.....	561
.....	561
line_profiler.....	562
cProfile.....	562
136:	564
.....	564
Examples.....	564
.....	564
.....	565
.....	566
.....	567
.....	567
while.....	568
137:	570

Examples.....	570
2.....	570
.....	571
138:	572
.....	572
.....	572
Examples.....	572
.....	572
.....	573
139:	575
Examples.....	575
.....	575
140:	576
.....	576
.....	576
Examples.....	576
.....	576
.....	577
.....	578
.....	578
Python 23six.....	578
.....	578
.....	579
.....	579
.....	579
.....	579
.....	580
141:	582
.....	582
Examples.....	582
.....	582
.....	

142:	584
.....	584
.....	584
Examples.....	584
.....	584
.....	586
.....	586
__all__.....	587
.....	588
.....	588
PEP8.....	589
.....	589
__import__.....	589
.....	589
Python 2.....	590
Python 3.....	590
143:	591
Examples.....	591
.....	591
.....	592
144:	593
.....	593
Examples.....	593
.....	593
unittest.mock.create_autospec.....	594
unittest.TestCase.....	595
.....	596
.....	597
pytest.....	598
145:	601
.....	601

Examples.....	601
.....	601
.....	601
.....	601
.....	601
randintrandrangerandomuniform.....	602
randint.....	602
randrange.....	602
.....	603
.....	603
.....	603
.....	604
.....	605
.....	606
146:	607
.....	607
.....	607
.....	607
Examples.....	607
.....	607
.....	609
.....	614
.....	614
.....	615
.....	615
.....	616
.....	616
.....	617
.....	618
.....	618
.....	619
.....	

147: **621**

..... 621

..... 621

Examples..... 621

3..... 621

..... 621

..... 621

..... 622

148: **623**

..... 623

..... 623

..... 623

Examples..... 623

..... 623

..... 625

..... 626

- 627

..... 627

2..... 628

149: **629**

..... 629

..... 629

..... 629

Examples..... 629

..... 629

else..... **630**

..... **630**

..... **631**

..... **632**

..... 632

.....

.....635

.....636

.....636

.....638

.....638

.....639

150: 640

Examples.....640

.....640

151: 641

Examples.....641

.....641

.....641

.....641

.....642

.....642

.....642

.....643

.....643

.....645

152: 646

.....646

Examples.....646

.....646

153: 650

Examples.....650

Python.....650

154: 651

.....651

.....651

.....

Examples.....	651
.....	651
For.....	652
.....	653
.....	653
break.....	653
continue.....	654
.....	654
returnbreak.....	654
else.....	655
.....	657
.....	657
While.....	658
.....	659
.....	659
.....	660
.....	660
"" do-while.....	661
.....	661
155:	663
Examples.....	663
Python.....	663
.....	664
156:	667
.....	667
Examples.....	667
locale.....	667
157:	668
Examples.....	668
.....	668

key.....	668
maxmin.....	668
.....	669
.....	669
.....	670
.....	670
.....	671
iterableNN.....	673
158:	675
.....	675
Examples.....	675
.....	675
.....	675
C-.....	676
PyPar.....	676
159:	678
.....	678
.....	678
Examples.....	678
.....	678
virtualenv	678
.....	678
.....	679
.....	679
.....	679
.....	679
.....	680
.....	680
Python.....	681
virtualenvwrapper.....	682
.....	682
.....	

.....683

.....684

Unix / LinuxPython.....684

virtualenv.....684

Anaconda.....685

.....685

.....685

.....686

.....686

.....686

160: 687

.....687

.....687

Examples.....687

.....687

.....687

.....688

.....688

.....689

.....689

.....691

.....692

.....693

.....693

.....694

.....**694**

.....**694**

.....695

161: 696

.....696

Examples.....696

PyTesseract.....	696
PyOCR.....	696
162: Python.....	698
Examples.....	698
IronPython.....	698
.....	698
.....	698
Jython.....	698
.....	699
.....	699
.....	699
.....	699
HTML.....	699
JavaScriptDOM.....	700
JavaScript.....	700
PythonJavaScript.....	701
.....	702
163:	703
.....	703
Examples.....	703
.....	703
.....	706
.....	707
.....	711
int.....	712
.....	713
sys.argv [0].....	713
h14.....	714
.....	714
GIL.....	715
.....	716

.....	716
PythonJSON.....	717
164:	718
.....	718
Examples.....	718
1n.....	718
.....	718
.....	721
.....	722
-	723
.....	723
165:	726
Examples.....	726
py2app.....	726
cx_Freeze.....	727
166:	729
.....	729
Examples.....	729
Python 2.43.3.....	729
.....	729
167:	730
.....	730
.....	730
.....	730
Examples.....	730
.....	730
.....	731
.....	731
.....	732
.....	733
.....	734
.....	734

.....	735
.....	735
.....	736
168:	737
Examples.....	737
.....	737
.....	738
169:	740
.....	740
Examples.....	740
.....	740
.....	741
.....	742
.....	742
.....	742
del.....	743
del v	743
del v.name	744
del v[item]	744
del v[a:b]	744
.....	745
.....	745
.....	745
.....	746
globalnonlocal Python 3	747
170:	749
.....	749
.....	749
.....	749
Examples.....	749
mapitertools.imapfuture_builtins.map.....	749

iterable.....	750
iterables.....	751
"None"Python 2.x.....	752
.....	753
171:	756
Examples.....	756
inputraw_input.....	756
.....	756
.....	757
.....	757
stdin.....	758
.....	758
172: intfloatstrtuplefrozensets.....	761
Examples.....	761
.....	761
.....	761
Frozenset.....	761
173:	762
Examples.....	762
.....	762
.....	764
174:	766
Examples.....	766
.....	766
.....	767
175:	768
.....	768
Examples.....	768
.....	768
.....	768
176:	771
Examples.....	771

.....	771
.....	771
.....	771
"__main__".....	772
177: abc	773
Examples.....	773
ABCMeta.....	773
/ ABCMeta@abstractmethod.....	774
178:	776
Examples.....	776
Python.....	776
179:	778
Examples.....	778
CHello World.....	778
C.....	779
C ++BoostC.....	779
C ++	779
180:	781
Examples.....	781
.....	781
.....	782
floortrunc.....	782
.....	782
.....	782
.....	783
.....	783
/.....	783
.....	783
.....	784
.....	784
.....	785
InfinityNaN.....	785

Pow.....	787
cmath.....	788
181:	791
.....	791
.....	791
.....	791
Examples.....	791
.....	791
.....	793
f-string.....	793
datetime.....	794
GetitemGetattr.....	794
.....	795
.....	795
.....	796
.....	797
.....	798
.....	798
Python 2.x.....	798
Python 3.2.....	798
.....	799
182: CSV	800
.....	800
.....	800
.....	800
Examples.....	800
.....	800
CSV.....	801
183:	802
.....	802
.....	803
Examples.....	803

.....	803
str.casefold().....	803
str.upper().....	803
str.lower().....	804
str.capitalize().....	804
str.title().....	804
str.swapcase().....	804
str.....	804
.....	804
str.split(sep=None, maxsplit=-1).....	805
str.rsplit(sep=None, maxsplit=-1).....	805
.....	806
str.replace(old, new[, count])	806
str.formatf-strings.....	806
.....	808
str.count(sub[, start[, end]]).....	808
.....	808
str.startswith(prefix[, start[, end]]).....	808
str.endswith(prefix[, start[, end]]).....	809
.....	809
str.isalpha.....	810
str.isupper str.islower str.istitle.....	810
str.isdecimal str.isdigit str.isnumeric.....	810
str.isalnum.....	811
str.isspace.....	811
str.translate.....	812
/.....	812
str.strip([chars]).....	812
str.rstrip([chars])str.lstrip([chars]).....	813
.....	813
1.....	814
.....	815
string.ascii_letters	815

string.ascii_lowercase	815
string.ascii_uppercase	815
string.digits	815
string.hexdigits	815
string.octaldigits	816
string.punctuation	816
string.whitespace	816
string.printable	816
.....	816
.....	817
strbytesunicode.....	817
.....	818
184:	820
Examples.....	820
2.....	820
datetime.....	820
datetime.....	820
185:	821
.....	821
Examples.....	821
datetime.....	821
.....	821
datetime.....	822
.....	822
datetime.....	823
.....	824
datetime.....	826
.....	826
ISO 8601.....	826
datetime.....	827
.....	827
.....	828
ISO 8601.....	828

.....	828
.....	828
.....	829
186:	830
.....	830
.....	830
Examples.....	830
ifelifelse.....	830
.....	830
if.....	831
Else.....	831
.....	831
.....	832
.....	832
.....	832
.....	833
.....	834
cmp2.....	834
.....	835
None.....	835
187:	837
Examples.....	837
.....	837
JPEG.....	837
188:	838
.....	838
.....	838
.....	838
.....	838
Examples.....	838
shelve.....	838

.....	839
.....	839
.....	840
189:	842
.....	842
Examples	842
str.indexstr.rindexstr.findstr.rfind	842
.....	842
.....	842
.....	843
.....	843
.....	843
Dict	843
list.indextuple.index	843
dict	844
bisect.bisect_left	844
.....	845
__contains__ iter__	846
190: PythonJavaScript	847
.....	847
Examples	847
`in`	847
191: CLI	848
.....	848
.....	848
Examples	848
.....	848
argparse	849
argparse	849
192:	852
.....	852
.....	852

Examples.....	852
.....	852
.....	853
.....	854
.....	855
.....	855
.....	855
.....	856
.....	856
.....	856
.....	856
.....	857
.....	857
.....	858
.....	858
.....	858
Flags.....	858
.....	859
`re.finditer`.....	859
.....	860
193:	861
.....	861
.....	861
Examples.....	861
.....	861
.....	862
.....	862
.....	863
.....	863
.....	863
`is`vs` == `.....	863
.....	

Common GotchaPython.....	865
194:	867
.....	867
.....	867
.....	867
Examples.....	867
.....	867
reduce.....	868
.....	868
/.....	868
/.....	869
195:	870
Examples.....	870
.....	870
.....	870
Itemgetter.....	870
196:	872
.....	872
.....	872
Examples.....	872
.....	872
next.....	872
.....	873
.....	873
.....	874
.....	876
.....	876
-	877
.....	878
.....	878
.....	879

.....	880
.....	880
.....	881
197:	882
.....	882
Examples	882
math.sqrtcmath.sqrt	882
**pow	883
mathmath.pow	883
math.expcmath.exp	884
1math.expm1	884
cmath	885
3pow	886
n	887
.....	887
198:	889
.....	889
Examples	889
.....	889
.....	890
199:	891
.....	891
.....	891
.....	891
Examples	891
.....	891
dict	892
KeyError	892
.....	893
.....	894
dict	894
.....	894

.....	894
dict().....	894
dict	894
.....	895
.....	895
**.....	896
.....	896
Python 3.5	896
Python 3.3	897
Python 2.x3.x	897
.....	897
.....	897
.....	898
.....	899
.....	899
200:	901
.....	901
Examples.....	901
IndentationErrorsSyntaxErrors.....	901
IndentationError / SyntaxError	901
.....	901
IndentationError / SyntaxError	902
.....	902
IndentationError	902
.....	902
IndentationError	902
.....	903
.....	903
TypeErrors.....	903
TypeError[/]	903
.....	903

TypeError[operand] '???' '???'	903
.....	904
TypeError '???'iterable / subscriptable	904
.....	904
TypeError '???'	904
.....	905
NameError '???'	905
.....	905
.....	905
import	905
PythonLEGB	905
.....	906
AssertionError	906
KeyboardInterrupt	906
ZeroDivisionError	907
.....	907
201:	909
.....	909
.....	909
.....	909
.....	909
Examples	909
.....	909
202:	911
.....	911
.....	911
Examples	911
.....	911
.....	912
append	913
insert	913

extendPython.....	913
fromlist.....	913
remove.....	913
pop.....	914
index.....	914
reversePython.....	914
buffer_info.....	914
count.....	914
tostring.....	915
tolistpython.....	915
fromstringchar.....	915
203:	916
Examples.....	916
Conda.....	916
204:	918
.....	918
.....	918
.....	918
.....	918
.....	919
Examples.....	919
.....	919
.....	921
.....	922
.....	922
.....	922
.....	922
.....	923
.....	923
.....	924
.....	924
.....	925

.....	925
.....	925
.....	926
.....	926
.....	926
/.....	927
.....	929
.....	930
.....	931
.....	932
.....	932
.....	933
.....	934
.....	935
.....	935
205:	937
.....	937
Examples.....	937
.....	937
206:	938
Examples.....	938
.....	938
.....	940

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

1: Pythonをいめる



Pythonはくわれているプログラミングです。それは

- ハイレベル Pythonはメモリなどのレベルをします。プログラマのがししかなくても、コードのやのコードなど、くのがあります。
- Pythonはすべてのコンテキストやでできるようにられています。のはPHPです。これは、にサーバーのWebスクリプトとしてされています。に、Python は、サーバーのWebだけでなく、デスクトップアプリケーションのにもできます。
- Pythonのすべてののは、あらゆるタイプのデータをできます。のは、なるになるタイプのデータにされることがあります。そのため、のコードがです

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

- くけされたプログラムのに、あなたがしているデータのとのないものはできません。たとえば、からへのれたはありません。からられたは、にしないりとしてわれません

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

- **Beginner friendly :) Pythonのとはにです。それはレベルであり、およびのでなプログラムをくことができるようにするためのをします。 Pythonは、オブジェクト、プログラミング、プログラミング、きプログラミングなど、のプログラミングパラダイムをサポートしています。それには、でなライブラリとインストールがなサードパーティのライブラリがあります。**

そのは[The Zen of Python](#)でされています。

、いくつかのきないがあるPythonの2つのなリリースブランチがあります。 Python 2.xはのバージョンですが、それでもなおくわれています。 Python 3.xでは、のをらすことをとしたののきわれています。どのバージョンがかをするには、[このをしてください](#)。

[のPythonのドキュメント](#)はかつなリソースであり、Pythonのすべてのバージョンのドキュメントと、めるためのチュートリアルもまれています。

にCPythonとばれるPython.orgによってされるのなと、のランタイムプラットフォームでのいくつかのがあります。 [IronPython](#) .NETプラットフォームでPythonを、 [Jython](#) Javaランタイム、

PyPy Pythonをのサブセットになどがあります。

バージョン

Python 3.x

バージョン	
[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	20110220
3.1	2009-06-26
3.0	2008-12-03

Python 2.x

バージョン	
2.7	2010-07-03
2.6	2008102
2.5	2006-09-19
2.4	2004-11-30
2.3	2003729
2.2	20011221
2.1	2001-04-15
2.0	2000-10-16

Examples

Pythonは、プログラミングのためにくされているなプログラミングで、Guido van Rossumによって1991年にリリースされました。Pythonはシステムとメモリをえ、オブジェクト、プログラミング、およびきスタイルがまれます。それはきくてなライブラリをっています。

Pythonの2つのメジャーバージョンがされています

- Python 3.xはのバージョンであり、です。
- Python 2.xはのバージョンであり、2020まではセキュリティアップデートのみをけります。しいはされません。くのプロジェクトでPython 2がされていますが、Python 3へのはになっています。

あなたは、ダウンロードやPythonのいずれかのバージョンをインストールすることができ、[ここ](#)で。Python 3とPython 2については、[Python 3とPython 2](#)をしてください。さらに、サードパーティのは、にされるライブラリやそののをして、データ、などののをにするPythonのパッケージをしています。[サイトのリスト](#)をしてください。

Pythonがインストールされているかどうかをする

Pythonがしくインストールされたことをするには、おみのでのコマンドをすることでできます。Windows OSをしているは、コマンドプロンプトでするにpythonのパスをするがあります。

```
$ python --version
```

Python 3.x 3.0

Python 3がインストールされていて、それがデフォルトのバージョンであるについては[トラブルシューティング](#)を、のようにされます。

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

Python 2.x 2.7

Python 2がインストールされていて、これがデフォルトのバージョンは[トラブルシューティング](#)をであれば、のようにされます。

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

Python 3をインストールしていて、`$ python --version`がPython 2バージョンをしている、Python 2もインストールされています。これはMacOSやくのLinuxディストリビューションでよくられます。Python 3インタプリタをにするには、わりに`$ python3`をします。

こんにちは、IDLEをったPythonの

IDLEはPythonのためのシンプルなエディタで、Pythonにバンドルされています。

IDLEで**Hello、World**プログラムをする

- したシステムでIDLEをします。
 - いバージョンのWindowsでは、[Windows]メニューの[All Programs]にあります。
 - Windows 8では、 `IDLE`するか、システムにするアプリでつけます。
 - UnixベースのシステムMacをむでは、 `$ idle python_file.py`してシェルからくことができます。
- にオプションがあるシェルがします。

シェルには、3つのプロンプトがあります。

```
>>>
```

に、プロンプトにのコードをします。

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

Enterをします。

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

Hello World Python ファイル

のをむしいファイル`hello.py`をします。

Python 3.x 3.0

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

Python 2.x 2.6

Python 2では、の`import`をしてPython 3の`print`をできます。

```
from __future__ import print_function
```

Python 2には、 [ここです](#)るように、 `__future__`モジュールをしてPython 3からオプションでインポートできるいくつかのがあります。

Python 2.x 2.7

Python 2をしているは、のをすることもできます。これはPython 3ではでないため、バージョンのコードがするため、されません。

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

で、 `hello.py` ファイルがあるディレクトリにします。

`python hello.py` と入力し、 `Enter` キーをします。

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

`Hello, World` がコンソールにされるはずです。

`hello.py` をファイルへのパスにきえることもできます。たとえば、ホームディレクトリにファイルがあり、Linuxでユーザーが "user" のは、 `python /home/user/hello.py` ます。

インタラクティブなPythonシェルをする

`python` コマンドをで `python`、なPythonシェルがされます。これは、 [Pythonインタプリタ](#) または REPL 'Read Evaluate Print Loop' ともされます。

```
$ 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
>>>
```

あなたのからPython 3をするには、 `python3` コマンドをします。

```
$ 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
>>>
```

あるいは、プロンプトをし、 `python -i <file.py>` ファイルをロードします。

コマンドラインでのコマンドをします。

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

Pythonシェルをじるにはのがあります

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

または

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

あるいは、`CTRL + D`はシェルをじて、ターミナルのコマンドラインにします。

Interpreterシェルのにあるに、にコマンドをキャンセルしてクリーンなコマンドプロンプトになるには、`CTRL + C`をします。

インタラクティブなPythonシェルをオンラインでしてみてください。

そののオンラインシェル

さまざまなウェブサイトがPythonシェルへのオンラインアクセスをしています。

オンラインシェルは、のにちます。

- Pythonのインストールがないマシンスマートフォン、タブレットなどからさなコードスニペットをします。
- なPythonをび、える。
- オンラインジャッジのをする。

はのリソースにはしません。

- <https://www.python.org/shell/> - のPythonウェブサイトがするオンラインのPythonシェル。
- <https://ideone.com/> - コードスニペットのをするために、ネットでくされています。
- <https://repl.it/languages/python3> - でシンプルなオンラインコンパイラ、IDE、インタプリタ。Pythonでコードを、コンパイル、します。
- https://www.tutorialspoint.com/execute_python_online.php - フルのUNIXシェルと、いやすいプロジェクトエクスプローラです。
- http://rextester.com//python3_online_compiler - をするシンプルでいやすいIDE

コマンドをとしてする

Pythonは、のコードをシェルのとしてすことができます

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

これは、シェルのスクリプトのをするにです。

シェルとそれをえて

パッケージ - PythonパッケージをインストールするためのPyPAツールは**PIP**です。インストールするには、コマンドラインで `pip install <the package name>` します。たとえば、 `pip install numpy` 。 Windowsでは、PATHにpipをするがあります。これをけるには、 `python -m pip install <the package name>` して `python -m pip install <the package name>` 。

シェル - これまで、Pythonのネイティブシェルをしてコードをするさまざまなについてしました。シェルはコードをリアルタイムでするためにPythonのをします。のシェルには、 **IDLE** にバンドルされたGUI、 **IPython** インタラクティブなエクスペリエンスをするためにられているなどがあります。

プログラム - のために、コンテンツを.pyファイルにし、シェル、 **IDE PyCharm** など、 **Jupyter ノートブック** などのツールをしてスクリプトやプログラムとして/することができます。ただし、ここでするは、するにはです。

Pythonチュートリアルでは、Pythonコードをステップすることで、プログラムのれをしたり、プログラムのどこがっているかをしたりすることができます。

PEP8は、Pythonコードをフォーマットするためのガイドラインをしています。コードのをえることはなので、コードのをすばやくみることができます。

のとのりて

Pythonでをするには、をしてをするだけです。

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

Pythonは=をとってにをします。にをするまたはデータをすることはありません。にをすると、がされ、そのでされます。にをせずにすることはありません。

```
# 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
```

のはからにします。したがって、のようになるとエラーがします。

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

Pythonのキーワードをなとしてすることはできません。あなたはキーワードのリストをのどることができます

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

の

1. は、またはアンダースコアでまるがあります。

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

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

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

2. のりのは、、、アンダースコアでされています。

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

3. はとがさ

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

Pythonでをするときデータをするはありませんが、になをメモリにりてると、Pythonインタプリタはにな **みみ** をします。

```

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'>

```

、あなたはりてのをっていますので、このようなところでPythonのについてえてください。

あなたがしている=をうには、のにあります=のオブジェクトのです。に、=うことはのをのオブジェクトのをりてるです。

あれは

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

たちがぶのであれば、のくのりてのから、`pi = 3.14`、そして`pi`オブジェクトにしてオブジェクトがのをつことができるので、ではなくである`3.14`。あなたがのをしていないなら、このにり、これをもうんでくださいまた、[このこと](#)をよりよくするためにることができます。

1にののをりてることができます。=のにじのがであることにしてください。

```

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
```

このエラーは、リネームをのことにすることでりくことができます。このダミーにはのけることができますが、なをりてるためにアンダースコア `_` をするのがです。

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

`_` のとりののはしくなければならないことにしてください。それは、のように `'` をアンパックするにはがすぎます'がスローされます。

```
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)
```

また、のに1つのをにりてることもできます。

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

このようなカスケードリテをする、`b`、`b`、`c` 3つの`a`すべてメモリのじオブジェクトをすることがです。が1の`int`オブジェクト。つまり、`a`、`b`、および`c`は3つのなる`int`オブジェクトにえられます。でのオブジェクトをりてると、りにのオブジェクトがされません。

```
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.
```

は、な `int`、`string`、`tuple`などのとじように、な `list`、`dict`などにもてはまります。

```
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]
```

ここまではですね。カスケードリテがなにされているときに、オブジェクトをすることになるとでったのオブジェクトにをりてるのとはに、はしなります。をて、あなたはそれをにるでしょう

```
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
```

れリストはPythonでもです。つまり、リストとしてのリストをめることができます。

```
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
```

に、Pythonのは、にしたのとじをとどめるはありません。しいをにするは、そのがなるであって
もに=をできます。

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

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

これがになるは、_のにあるものはオブジェクトのにぎないというについてえてみてください。
まず、あなたがびす`int2`をつオブジェクト、あなたはあなたのをし、えることをし`a`にstring「
しい」をつ、オブジェクト。シンプル、そう

ユーザ—

インタラクティブ

ユーザからのをするには、`input`をし`input` Python 2.xでは、わりに`raw_input`とばれますが、
Python 2.xではになるのバージョンの`input`があります。

Python 2.x 2.3

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

セキュリティ Python2で`input()`をしないでください - `input()`されたテキストはPython
のPython3の`eval(input())`とじようにされます。このをするリスクのについては、[こ
の](#)をしてください。

Python 3.x 3.0

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

こののりのでは、Python 3のをします。

このはをります。はプロンプトとしてし、をします。のコードは、ユーザがをつプロンプトをし
ます。

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

ユーザーが「Bob」としてEnterキーをすと、nameが"Bob"りてられます。

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

inputはにstrであることにしてください。これは、ユーザーがをしたいにです。したがって、と
してするにstrをするがありstr

```
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
```

try / except ブロックをして、ユーザーのをするにをキャッチすることをおめします。えは、あな
たのコードがraw_inputをint raw_inputにキャストraw_inputうとしていて、ユーザがきむものがキ
ャストできないValueError、ValueErrorます。

IDLE - Python GUI

IDLEはPythonのであり、コマンドラインにわるものです。がすように、IDLEはしいコードのや
Pythonのににちます。WindowsではこれがPythonインタプリタにしていますが、のオペレーティ
ングシステムでは、パッケージマネージャをしてインストールするがあります。

IDLEのなはのとおりです。

- の、およびスマートインデントきのマルチウィンドウテキストエディタ
- シンタックスハイライトきのPythonシェル
- ステッピング、ブレークポイント、およびコールスタックをえたデバugg
- インデントがPythonのインデントについてぶのに
- Pythonプログラムを.pyファイルとしてしてし、でIDLEをしてそれらをします。

IDLEでは、F5かrun Python Shellをrun Python Shellでインタプリタをします。IDLEをすると、ユ
ーザーがきむときにコードがされるため、しいユーザーにとってよりれたになります。

がたくさんあることにしてください。たとえば、[このディスカッション](#)や[このリスト](#)をしてくだ
さい。

トラブルシューティング

• Windows

Windowsの、デフォルトのコマンドは`python`です。あなたがけるは"`'python' is not recognized`"エラーが、ものいは、Pythonのは、おいのシステムのにはないということです `PATH`。これは、[マイコンピュータ]をクリックして[プロパティ]をするか、[コントロールパネル]から[システム]にしてアクセスできます。「なシステム」をクリックし、に「...」をクリックします。 `PATH`をしてPythonインストールのディレクトリとScriptフォルダは `C:\Python27;C:\Python27\Scripts` を `C:\Python27;C:\Python27\Scripts` ます。これにはがであり、がながあります。

じマシンでのバージョンのPythonをする、は `python.exe` ファイルのをすることです。たとえば、あるバージョンの `python27.exe` を `python27` すると、 `python27` はそのバージョンのPythonコマンドになります。

また、Python Launcher for Windowsをすることもできます。これはインストーラからでき、デフォルトでされます。 `python[xy]` わりに `py -[xy]` をってするPythonのバージョンをすることができます。のバージョンのPython 2をするには、 `py -2` でスクリプトをし、 `py -3` スクリプトをしてのバージョンのPython 3をします。

• Debian / Ubuntu / MacOS

このセクションでは、 `python` ファイルのが `PATH` にされていることをとしています。

Debian / Ubuntu / MacOSをしているは、をいて、Python 2.xでは `python`、Python 3.xでは `python3` します。

し `which python` Pythonインタプリタがされるかをします。

• アーチLinux

アーチのLinuxおよびのデフォルトのPythonは、Python 3であるので、 `python` か `python3` Pythonの3.xおよび `python2` のPython 2.xの

• のシステム

Python 3は `python3` わりに `python` にバインドされることがあります。Python 2がインストールされているこれらのシステムでPython 2をするには、 `python2` をできます。

データ

ビルトインタイプ

ブール

`bool` `True`または`False`ブール。のような`and`、`or`、`not`ブールですることができます。

```
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
```

Python 2.xとPython 3.xでは、`boolean`も`int`です。 `bool`は`int`のサブクラスであり、`True`および`False`はそのインスタンスです。

```
issubclass(bool, int) # True

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

ブールがでされる、`True`および`False`は1と0 がをすためにされます。

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

- `int`

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

Pythonのはのサイズです。

Pythonのいバージョンでは、`long`がで、これは`int`とはなります。 2つはされています。

- `float`。 はとシステムアーキテクチャにしますが、CPythonでは`float`データがC `double`にします。

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

- `complex`

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

`<`、`<=`、`>`および`>=`は、オペランドがであるに`TypeError`を`TypeError`ます。

Python 3.x 3.0

- `str`

ユニコード。 `'hello'` のタイプ

- `bytes` バイト。 `b'hello'` のタイプは、

Python 2.x 2.7

- `str` バイト。 `'hello'` のタイプ
- `bytes str`
- `unicode` ユニコード。 `u'hello'` のタイプは、

とコレクション

Pythonは、けされたシーケンスとけられていないコレクション `set` や `dict` をします。

- `str`、`bytes`、`unicode` はシーケンスです
- `reversed reversedstr reversed`

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

- `tuple` のタイプの`n`のけられたコレクション `n >= 0`。

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

インデックスをサポートします。;すべてのメンバーがハッシュなハッシュ

- `list` `n` のけられたコレクション `n >= 0`

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

ハッシュではありません。です。

- `set` ののけられていないコレクション。アイテムはハッシュでなければなりません。

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

- `dict` ユニークなキーとのペアのけられていないコレクション。キーはハッシュでなければなりません。

```
a = {1: 'one',
     2: 'two'}

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


オブジェクトは、それがそれがそののにしないハッシュをする、ハッシュである `__hash__()` メソッドを、そしてそれがどのオブジェクトとすることができる `__eq__()` メソッド。をするハッシュオブジェクトは、じハッシュをつがあります。

みみ

みみデータにして、ビルトインにはのみみがあります。

- `True` みみ`bool`のの
- `False` みみ`bool`の`false`
- `None` がしないことをするためにされるシングルトンオブジェクト。
- `Ellipsis` または... Python3 +のとなるでされ、のとしてPython2.7 +でのがされています。
`numpy` やそれにするパッケージはこれをの 'すべてをむ' としてします。
- `NotImplemented` なメソッドがのをサポートしていないことをPythonにすためにされるシングルトン`NotImplemented` はなにはメソッドをします。

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

Python 3.x 3.0

`None` のけがありません。 `<`、`<=`、`>=`、`>` をすると、これサポートされず、`TypeError` ます。

Python 2.x 2.7

`None` はにのよりさくなります `None < -32` は`True` され`True` 。

ののテスト

Pythonでは、みみの`type` をってオブジェクトのデータをべることができます。

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

では、`isinstance` を`isinstance` でデータをテストすることができます。しかし、のにすることはされません。

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

`type()` と `isinstance()` については、Pythonの`isinstance`と`type`のいをしてください。

かが`NoneType`かどうかをテストするには

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

データの

なデータをできます。

たとえば、`'123'`は`str`であり、`int`をしてにできます。

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

`'123.456'`のようなからの、`float`をしてうことができます。

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

シーケンスまたはコレクションのタイプをすることもできます

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

リテラルにおけるな

のすぐに1のラベルをけると、どのタイプのをしたいのかをすることができます。

- `b'foo bar'` Python 3の`bytes`、Python 2の`str`
- `u'foo bar'` Python 3の`str`、Python 2の`unicode`
- `'foo bar'` `str`
- `r'foo bar'` のとばれる。エスケープするはありません。したとおりにすべてがそのままきがれます。

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

およびなデータ

なオブジェクトは、 とされます。たとえば、あるにリストをすと、リストをすることができます。

```
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
```

どのようなでもできないオブジェクトはとされます。たとえば、をすることがないため、はです。

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

はであるため、`x`をすることができますが、これは`x`がにしたオブジェクトをするものではありません。`x`はしいオブジェクトをしすだけでした。

インスタスがなデータは、なデータとされ、のオブジェクトおよびデータもです。

データの

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

データの

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

みみのモジュールと

モジュールは、Pythonのとをむファイルです。は、あるロジックをするコードです。

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

Pythonのみみをべるには、`dir()`うことができます。`dir()`なしでびされたは、のスコープのをします。そうでないは、されたオブジェクトのとそのからなをむのアルファベットリストをします。

```
>>> 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'
]
```

のをるために、みみの`help`することができます。

```
>>> 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.
```

みみのモジュールにはながまれています。えは、`math`をめるには、`math`モジュールをめるがあります。

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

モジュールのすべてののをるために、リストをにしてをすることができます。

```
>>> 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',
'loglp', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'trunc']
```

`__doc__`は、にいくつかのドキュメントをするのにだとわれます

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

にえて、ドキュメントもモジュールですることができます。したがって、 `helloWorld.py` というのファイルがあるは、 `helloWorld.py` ようにします。

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

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

のようにdocstringにアクセスできます

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

- のユーザ、その、そのクラスの、およびそのクラスのクラスのをにするには、`dir`をします。

```
>>> 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__']
```

のデータは、 `str` というみみをしてににでき `str`。これは、データが `print` されたときにデフォルトでひされます

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

ブロックインデント

Pythonはインデントをしてコントロールとループのをします。これはPythonのにしますが、プログラマはののにのうがあります。したがって、エディタのによってしないをするコードがするがあります。

Pythonはコロンしています：コードのブロックのとをすためにインデントをあなたがのからたは、らかのにしているとしないでください。つまり、、ループ、 `if`、そののなど、Pythonのブロックにはがありません。すべてのブロックはコロンでまり、そのにインデントされたがまれます。

例えば

```
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
```

または

```
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
```

に1つのステートメントをむブロックは、じにくことができますが、このにはいスタイルとはみなされません。

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

1つのステートメントでこれをしようとしてもしません。

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

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

のブロックは、`IndentationError`きこします。コンテンツをたないブロックがあるときは`pass` もしないコマンドをう

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

スペースとタブ

つまり、インデントにはに 4つのスペースをします。

にタブをすることはですが、Pythonコードのスタイルガイドである[PEP 8](#)では、スペースがされることがされています。

Python 3.x 3.0

Python 3ではインデントにタブとスペースをさせることはできません。このような、コンパイルにエラーが `Inconsistent use of tabs and spaces in indentation` されず、プログラムがされません。

Python 2.x 2.7

Python 2ではインデントでタブとスペースをさせることができます。これはくおめします。タブは、のインデントを **8のスペースにします**。エディタはタブを **4つのスペースの**としてするよう

にされていることがであるため、なバグがするがあります。

PEP 8

`-t` オプションをしてPython 2コマンドラインインタプリタをびすと、タブとスペースをにせるコードにするがされます。 `-tt` を `-ttt` これらのはエラーになります。これらのオプションはくおめします

くのエディタでは、「タブをスペースに」しています。エディタをするときは、タブ `\t` と `Tab` キーをするがあります。

- なくとも、なインデントがなは、セマンティクスとするように、タブを8つのスペースをするようにするがあります。はタブをスペースににすることもできます。
- ただし、 `Tab` キーをすとタブをするわりに、4つのスペースをするようにエディターをするとです。

タブとスペースをさせていたPythonソースコード、またはのインデントスペースをとってかれたPythonソースコードは、`autopep8` をって `pep8` にさせることができます。それほどでないはほとんどのPythonインストールにあります `reindent.py`

コレクションの

Pythonにはいくつかのコレクションがあります。 `int` や `str` などののはのをしますが、コレクションはのをします。

リスト

おそらく `list` はPythonでもにわれるコレクションです。そのにもかかわらず、リストはの、にJavaScriptのによくています。Pythonでは、リストはになPythonのけられたコレクションです。カンマでってをカッコでむことで、リストをできます。

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

リストはにすることができます

```
empty_list = []
```

リストのはのデータにされているわけではありません。Pythonがであるとすればがあります。

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

リストには、そのとしてのリストをめることができます。

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

リストのは、インデックスまたはそののをしてアクセスできます。Pythonのリストはゼロインデックスされているので、リストののはインデックス0に、2のはインデックス1に、というようにきます。

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

インデックスは、リストのからえることをするのでもかまいません `-1`はののインデックスです。したがって、ののリストをすると

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

リストはなので、リストのをすることができます

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

また、リストからをおよび/またはすることもです。

`L.append(object)` でリストののにオブジェクトをし、`None`します。

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

のインデックスにリストするしいをします。 `L.insert(index, object)`

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

`L.remove(value)` を `L.remove(value)` てにれるをし、`None`します。

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

が`x`であるののリストのインデックスをします。そのようながない、エラーをします。

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

リストのカウントのさ

```
len(names)
6
```

リストのものをカウントする

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

リストをする

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

`L.pop([index])` してインデックスのをしてしますデフォルトはの。

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

のようなリストをできます

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

タプル

`tuple` はリストにいますが、です。そのため、タプルのをすることも、をタプルにすることもタプルからすることもできません。タプルは、IPアドレスやポートなど、するのしないさなのコレクションによくされます。タプルはカッコのわりにかっこでされます。

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

リストのじけルールもタプルにされます。タプルもれにすることができ、はなのなPythonにすることができます。

1つのメンバしかたないタプルをするがありますカンマにしてください。

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

または

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

またはに `tuple` をする

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

Pythonの `dictionary` は、キーとのペアのです。はでまれています。ペアはコンマでられ、キーと

はコロンでられます。にをします。

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

をするには、キーでしてください

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

のすべてのキーをしてからりしてもかまいません。

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

はJSONによくています。 Pythonライブラリのネイティブ `json` モジュールをして、JSONとをすることが出来ます。

セット

`set` とは、りしはなく、ではなくソートののコレクションです。それらは、いくつかのものがグループされていることだけがあり、それらがどのようなでまれているのではないというでされます。なデータグループの、が `set` にあるかどうかをチェックするが、 `list` にしてじことをうよりもはるかにです。

`set` をすることは、 `dictionary` をすることとよくてい `dictionary` 。

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

または、の `list` をして `set` をすることもできます。

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

`in` を `in` て `set` メンバーシップをチェックしてください

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

リストとまったくじ `set` をできますが、はののになります。

defaultdict

`defaultdict` はキーのデフォルトをつで、にがされていないキーにはエラーなしでアクセスできます。 `defaultdict` は、しいキーがされるたびに作るがないというで、のがコレクションリスト、な

どであるににです。

`defaultdict` は `KeyError` をさせません。しないキーは、されるデフォルトをします。

たとえば、のをえてみましょう。

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

しないキーにアクセスしようとすると、pythonはのようエラーをします

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

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

KeyError: 'Alabama'
```

`defaultdict` してみましょう。これはコレクションモジュールにあります。

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

えられたキーがしないにデフォルト **Boston** をすることです。とじようにdictをします。

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

しないキーでdictにアクセスしようとすると、pythonはデフォルト、すなわちBostonをします

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

のキーのしたをの `dictionary` にします

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

ヘルプユーティリティー

Pythonにはインタプリタにみまれたいくつかのがあります。キーワード、みみ、モジュール、またはトピックのをするには、Pythonコンソールをき、のようします。

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

キーワードをしてをけります

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

またはユーティリティで

```
help> help
```

をします

```
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)
```

モジュールのサブクラスをリクエストすることもできます

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

ヘルプをして、インポートしたさまざまなモジュールのドキュメントストリングにアクセスできます。たとえば、のようにします。

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

あなたはエラーがします

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

そして、モジュールでなメソッドのリストをしますが、インポートしたでなければなりません。

quitでヘルパーをじる

モジュールの

モジュールは、とステートメントをむインポートなファイルです。

モジュールは、.py ファイルをすることでできます。

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

モジュールのは、モジュールをインポートすることによってできます。

したモジュールのは、インポートするファイルとしディレクトリにするがあります。ただし、あらかじめみまれているモジュールをPythonのlibディレクトリに入れることもできますが、であればけるべきです。

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

モジュールはのモジュールによってインポートできます。

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

モジュールののをインポートすることができます。

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

モジュールにエイリアスをけることができます。

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

モジュールはスタンドアロンのなスクリプトにすることができます。

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

それをします

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

モジュールがディレクトリにあり、pythonでするがある、ディレクトリには__init__.pyというのファイルがまれているがあります。

- strおよびrepr

オブジェクトのみなをするためにできる2つのがあります。

repr(x)はx.__repr__()をします。x.__repr__() x° evalはこののをオブジェクトにします。

str(x)はx.__str__() str(x)をします。オブジェクトをするがなです。これにより、ながされるがあります。

repr

くのでは、このはeval()にされるときにじをつオブジェクトをするをそうとします。そうでない、は、のとにオブジェクトのタイプのをむでまれたです。これには、オブジェクトのとアドレスがまれることがよくあります。

str

のは、をします。thisとrepr(object)のいは、str(object)はずeval()けれられるをそうとはしないということです。むしろ、そのは、な、つまり「がめる」をすことです。がされていない、の、がされます。

1

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

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)'
```

クラスをするときに、これらのメソッドをオーバーライドして、なをうことができます。


```
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)
```

のクラスをして、をできます。

```
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
```

pipをしてモジュールをインストールする

pipは、pythonパッケージインデックスPyPIでなからかパッケージをインストールするがあるときにあなたのです。python.org>からダウンロードしたPython 2> = 2.7.9またはPython 3> = 3.4をしている、pipはすでにインストールされています。Linuxまたはネイティブパッケージマネージャーをえたの* nixをするコンピューターでは、pipでインストールするがあります。

インストールのPython 2とPython 3のをつインスタンスでは、pipくの、Pythonの2とをしpip3つてPython 3にpipのPython 2とのパッケージをインストールしますpip3だけのPython 3のパッケージをインストールします。

パッケージのインストール

パッケージをするのは、とじくらいです

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

パッケージをインストールするのは、タイピングとじくらいです/コマンドプロンプトで、Pythonインタプリタではない

```
$ 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
```

`xxx`は、インストールするパッケージのバージョンです。

サーバーがプロキシのろにある、のコマンドをしてパッケージをインストールできます。

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

インストールされたパッケージのアップグレード

インストールされているパッケージの新しいバージョンがされたら、システムにインストールされません。インストール済みのパッケージのうち、どれがになっているかのをるには、のコマンドをします。

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

のパッケージのをアップグレードする

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

いパッケージをすべてすることは、`pip`ではありません。

ピップをアップグレードする

のコマンドをしてのpipインストールをアップグレードできます

- LinuxまたはMacOS Xの

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

いくつかのLinuxシステムでは、`pip`で`sudo`をうがあるかもしれません

- Windowsの

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

または

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

ピップにする[はこちらをしてください](#)。

Python 2.7.xおよび**3.x**のインストール

のは、Python 2.7にかれていすがないり。Python 3.xのはています。

WINDOWS

まず、Python 2.7のバージョンをWebサイト <https://www.python.org/downloads/>からダウンロードしてください。バージョンはMSIパッケージとしてされます。でインストールするには、ファイルをダブルクリックしてください。

デフォルトでは、Pythonはディレクトリにインストールされます

```
C:\Python27\
```

インストールはPATHをにしません。

あなたのPythonのインストールがC\ Python27にあるとし、これをPATHにしてください

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

Pythonのインストールがcmdでかどうかをする

```
python --version
```

Python 2.xおよび3.xのサイドバイサイド

WindowsマシンにPython 2.xと3.xをしてインストールしてするには

1. MSIインストーラをしてPython 2.xをインストールします。

- Pythonがすべてのユーザーにインストールされていることをしてください。
- オプションPythonをPATHにして、pythonをしてPython 2.xをコマンドラインからびしにします。

2. それぞれのインストーラをしてPython 3.xをインストールします。

- ここでも、すべてのユーザーにPythonがインストールされていることをしてください。
- オプションPythonをPATHにして、Python 3.xをpythonをとってコマンドラインからびしにします。これはPython 2.xのPATHよりもされるがありますので、PATHをしてをしてください。
- すべてのユーザーにpy launcherをインストールしてください。

Python 3は、Python 2.xとPython 3.xをコマンドラインからにするためにできるPythonランチャーをインストールします。

```
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.
>>>
```

のPythonバージョンにするpipバージョンをするには、のようになります。

```
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

CentOS、Fedora、Redhat EnterpriseRHEL、UbuntuのバージョンにはPython 2.7がしています。

Python 2.7をLinuxにインストールするには、のようになります

```
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
```

また、PATHにしいpythonのパスをします。しいPythonが/root/python-2.7.X、 export PATH = \$PATH:/root/python-2.7.Xしexport PATH = \$PATH:/root/python-2.7.X

Pythonのインストールがかどうかをターミナルでする

```
python --version
```

Ubuntuソースから

Python 3.6がなは、にすようにソースからインストールできますUbuntu 16.10および17.04は Universal Repositoryに3.6バージョンがあります。 Ubuntu 16.04のバージョンでは、のにわなければなりません。

```
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
```

マックOS

たちがうように、macOSはPython 2.7.10でインストールされていますが、このバージョンはく、のPythonからされています。

OS XにされているPythonのバージョンはにはですが、にはいていません。OS Xにされているバージョンは、したプロダクションバージョンとみなされているのPythonのリリースよりいかもしれません。 [ソース](#)

[ホームブリュー](#)をインストールする

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

Python 2.7をインストールする

```
brew install python
```

Python 3.xでは、わりに`brew install python3`コマンドをします。

オンラインでPythonをいめるをむ <https://riptutorial.com/ja/python/topic/193/python>をいめる

2: "pip"モジュールのPyPI Package Manager

き

Pythonのpipパッケージマネージャをうがえるかもしれません。のインポートで`ImportError`がし、そのをしたい。Windowsで`Python_root/Scripts/pip.exe`をすると、pipパッケージの`main`クラスがインポートされる`__main__.py`ファイルがされます。つまり、pipファイルをするたびにpipパッケージがされます。ファイルとしてpipをするには、pip [PyPI Package Manager](#)をしてください。

- pip。 <function | attribute | class> functionはのいずれかです。
 - オートコンプリート
 - メインオプションパーサーおよびオプションとそのサブコマンドおよびオプションのコマンドとオプションの。シェルスクリプトbash、zsh、fishのいずれかをソースとしてにします。
 - `check_isolatedargs`
 - param args {list}
 - returns {boolean}
 - `create_main_parser`
 - `{pip.baseparser.ConfigOptionParserオブジェクト}`
 - `mainargs = None`
 - param args {list}
 - returns {integer}しなかったは0をします。
 - `parseoptsargs`
 - param args {list}
 - `get_installed_distributions`
 - {list}をす
 - `get_similar_commandsname`
 - コマンドはされます。
 - パラメータ{string}
 - returns {boolean}
 - `get_summariesordered = True`
 - ソートされたコマンド、コマンドタプルをします。
 - `get_prog`
 - {string}をします。
 - `dist_is_editable`
 - ディストリビューションはなインストールですか
 - param dist {object}
 - returns {boolean}
 - `commands_dict`
 - {}

Examples

コマンドの

```
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])
```

`pip.main(['freeze'])`と`pip.main(['freeze', '', ''])`がです。

バッチインストール

1のびしでのパッケージをすことはですが、1つのインストール/アップグレードがすると、インストールプロセスがし、ステータスが'y'でします。

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

いくつかのインストールがしたときにしたくないは、ループでインストールをびします。

```
for i in installed:
    pip.main(['install']+i.key+['--upgrade'])
```

ImportErrorの

モジュールとしてPythonファイルをする、パッケージがインストールされているかどうかはずしもするはありませんが、それでもスクリプトにはです。

```
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
```

インストール

バージョン3.4のようなくのパッケージは3.6でもうまくしますが、のプラットフォームのディストリビューションがないはインストールできませんが、があります。 .whlファイルホイールと呼ばれるのでは、のプラットフォームにパッケージをインストールできるかどうかをします。えは。

scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl [scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl] - [バージョン] - [Pythonインタプリタ] - [Python-interpreter] - [オペレーティングシステム] .whl。ホイールファイルのがされた、プラットフォームがするので、pipはプラットフォームまたはPythonのバージョンがしなくてもパッケージをインストールしようとしています。からプラットフォームまたはインタプリタをすると、pipモジュールkjhfkdjdf.whl is not a valid wheel filename.のバージョンでエラーがしkjhfkdjdf.whl is not a valid wheel filename.。

わりに.whlファイルは7-zipとしてアーカイバをしてすることができます。 - 、メタフォルダとソースファイルをむフォルダがまれています。これらのソースファイルは、このホイールにインストールスクリプトがまれていないり、にsite-packagesディレクトリにすることができます。もしそうなら、にするがあります。

オンラインで"pip"モジュールのいPyPI Package Managerをむ

<https://riptutorial.com/ja/python/topic/10730/-pip-モジュールのい-pypi-package-manager>

3: * argsと** kwargs

すべきがいくつかあります

1. argsとkwargsというはでされていますが、のではありません。したがって、これらはです

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

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

2. のargsまたはのkwargsパラメータをつことはできませんただしではありません

```
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. *argsにくのは、だけですことができるキーワードのみのです。*argsわりにのをして、リストをせずにをキーワードにすることができます。キーワードのみのパラメータリストは、Python 3でのみできます。

```
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はパラメータリストのになければなりません。

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

Examples

をくときに*** args**をう

タブルのすべてのつまりのないをするをくときに*をうことができます

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

びしメソッド

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

そのびしでは、**farg**はいつものようにりてられ、の2つはけったに**args**タブルにされます。

をくときに**** kwargs**をう

パラメータのにダブルスター**をして、ののキーワードきをるをすることができます。

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

メソッドをびすと、Pythonはすべてのキーワードのをし、それをでできるようにします

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

の**** kwargs**パラメータはにのパラメータでなければならず、のパラメータのにされたにのみすることにしてください。

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

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

ので、**kwargs**はとじでされます。 **kwargs** 々のにアクセスするには、のと同じようにループします。

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

`print_kwargs(a="two", b=1)` をびすと、のがされます。

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

をびすときに *** args** をう

の ***args** なは、をラップまたはしたのいずれかにすることです。なはクラスの `__init__` メソッドにあるかもしれません

```
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
```

ここで、`a` パラメータは、のすべてのおよびキーワードがクラスにされ、されたに、クラスによってされます。

えは

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

ここでこるのは、クラス `B` の `__init__` が `1, 2, 3` することです。1つの `a` をるがあることをっているので、`1` でされたのをするので、`a == 1` スコープでします。

に、の `*args` をるがあるので、`1, 2` でされたりのをり、`*args` ます。ので `args == [2, 3]` 。

に、`*args` ってクラス `A` の `__init__` をびします。Pythonは `args` のに `*` をて、リストをに `""` します。クラス、このでは、`B` の `__init__` はクラス `A` の `__init__` は、がされます `2, 3` すなわち、`A(2, 3)`

に、の `x` プロパティを `1` にしいの `a` にします。

をびすときに **** kwargs** をう

ディクショナリをして、のパラメータにをりてることができます。のキーとしてパラメータ `name` をし、キーにバインドされたこれらののをします。

```
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)
```

をびすときに* argsをう

をびすときにに*をすると、リストまたはタプルをアンパックするがあります

```
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
```

スターきのさは、ののと同じであるがあることにしてください。

なPythonのイディオムは、zipでアンパック*をしてそのをさせることです

```
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)
```

キーワードのみとキーワードがな

Python 3では、デフォルトをしなくてもキーワードでしかできないをすることができます。これはstar *をして、キーワードパラメータをせずにのパラメータをすることによってわれます。*ののすべてののは、キーワードのみのつまり、のです。キーワードのみにデフォルトがえられていないは、をびすときにききであることにしてください。

```
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
```

kwargsをにれる

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

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

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

**kwargsとデフォルト

****kwargs**でデフォルトをするには

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

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

オンラインで* argsと** kwargsをむ <https://riptutorial.com/ja/python/topic/2475/--argsと---kwargs>

4: `__name__`

き

`__name__` は、ファイルがモジュールとしてインポートされたかどうかをチェックし、`__name__` で、クラス、モジュールオブジェクトをするためにされます。

Pythonの`__name__` は、そのモジュールをむがされています。トップレベルインタラクティブなインタプリタやメインファイルなどでは、`'__main__'` されています。これは、モジュールがインポートされるのではなくされているに、のブロックをするためにできます。

するな `obj.__name__` は、クラス、インポートされたモジュール、メソッドをむでつけられ、されたときにオブジェクトのをえます。

Examples

```
__name__ == '__main__'
```

な `__name__` はユーザによってされていません。に、モジュールがされているかどうか、または `import` がされたためにされているかどうかをチェックするためにされます。モジュールがインポートされるときにコードのをし `if __name__ == '__main__':` にするには `if __name__ == '__main__':` か `if __name__ == '__main__':` します。

module_1.pyを 1にする

```
import module2.py
```

そして、がこるのかを、 **module2.py**にじててみましょう

1

module2.py

```
print('hello')
```

module1.pyをすると `hello` されます

module2.pyをすると `hello` されます

2

module2.py

```
if __name__ == '__main__':  
    print('hello')
```

module1.pyをすることもされません
module2.pyをすると`hello` されます

function_class_or_module.__name__

、クラスまたはモジュールのな`__name__`は、そのをむです。

```
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
```

ただし、`__name__`は、クラス、メソッド、またはををするのではなく、にえられたです。

```
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
```

これは、とりわけ、デバッグのためにすることができます

```
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
```

ロギングでの

みみ `logging` をするののパターンは、のモジュールの `__name__` でロガーをすること `__name__` 。

```
logger = logging.getLogger(__name__)
```

つまり、モジュールのがログにされ、メッセージがどこからたのかがわかりやすくなります。

オンラインで `__name__` をむ <https://riptutorial.com/ja/python/topic/1223/--name-->

5: `exec`と`eval`を使ったコード

- `eval[、globals = None [、locals = None]]`
- `exec`オブジェクト
- `exec`オブジェクト、グローバル
- `exec`オブジェクト、グローバル、ローカル

パラメーター

expression	としてのコード、または <code>code</code> オブジェクト
object	としてのステートメントコード、または <code>code</code> オブジェクト
globals	グローバルにする。 <code>locals</code> がされていない、これはの々にもされます。すると、 <code>globals()</code> がされます。
locals	ローカルにされるマッピングオブジェクト。すると、わりに <code>globals</code> されたものがされます。をすると、 <code>globals()</code> および <code>locals()</code> がそれぞれ <code>globals</code> および <code>locals</code> され <code>globals</code> 。

`exec`では、`globals`が`locals`つまり、`exec`オブジェクトをする、コードはモジュールレベルにあるかのようにされます。`globals`と`locals`がなるオブジェクトである、コードはクラスにあるかのようにされます。

`globals`オブジェクトがされても`__builtins__`キーがされていない、Pythonのみみとはにグローバルスコープにされます。されたスコープでの`print`や`isinstance`などののをするには、`globals``__builtins__`というキーを`None`マップさせます。ただし、これはセキュリティではありません。

Python 2のはしないでください。 Python 3のはPython 2でします。したがって、のはされません
<S>

- `exec` object
- `exec` object in `globals`
- `exec` object in `globals`, `locals`

Examples

`exec`での

```
>>> code = """for i in range(5):\n    print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
```

evalでをする

```
>>> expression = '5 + 3 * a'
>>> a = 5
>>> result = eval(expression)
>>> result
20
```

をプリコンパイルしてする

`compile`みみをして、をコード・オブジェクトにプリコンパイルすることができます。このコードオブジェクトを`eval`にすことができます。これは、されたコードのりしをスピードアップします。`compile`する3のパラメータは、`'eval'`があります。

```
>>> 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
```

カスタムグローバルをしてevalでをする

```
>>> variables = {'a': 6, 'b': 7}
>>> eval('a * b', globals=variables)
42
```

プラスとして、このコードでは、ってでされたをすることはできません。

```
>>> 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
```

`defaultdict`すると、たとえばをゼロにすることができます。

```
>>> from collections import defaultdict
>>> variables = defaultdict(int, {'a': 42})
>>> eval('a * c', globals=variables) # note that 'c' is not explicitly defined
0
```

ast.literal_evalをしてPythonリテラルをむをする

、などのPythonリテラルをむがある、`ast.literal_eval`をして`eval`ではなくそのをできます。これには、ののみをするというがあります。

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

しかし、これはできないユーザによってされるコードのにはではなく、にされたでをクラッシュさせるのはではありません

```
>>> import ast
>>> ast.literal_eval('(' * 1000000)
[5] 21358 segmentation fault (core dumped) python3
```

ここで、は100りされた()で、CPythonパーサーでクラッシュをきこします。CPythonは、パーサのバグをセキュリティのとはみなしません。

exec、**eval**、**ast.literal_eval**をしてできないユーザーがするコードをする

できないユーザーのコードをにするために**eval**や**exec**をすることはできません。`ast.literal_eval`でもパーサーでクラッシュするがあります。のあるコードのをぐことは々ありますが、パーサーやトークナイザでのクラッシュのをするものではありません。

できないユーザーがコードをするには、サードパーティのモジュールにをけるか、のパーサーとのマシンをPythonでするがあります。

オンラインで`exec`と`eval`をつたコードをむ <https://riptutorial.com/ja/python/topic/2251/-exec-と--eval-をつたコード>

6: 2to3 ツール

- \$ 2to3 [-options] path / to / file.py

パラメーター

パラメータ	
ファイル/ディレクトリ	2to3は、としてされるファイルまたはディレクトリのリストを けます。ディレクトリは、Pythonソースのためににトラバー スされます。
オプション	オプションの
-f FIX、--fix = FIX	するをします。デフォルトall。 --list-fixesなをリスト--list- fixes
-j PROCESSES、-- processes = PROCESSES	2to3をに
-x NOFIX、--nofix = NOFIX	トランスフォーマーションをする
-l、--list-fixes	なのリスト
-p、--print-function	print()がとなされるようにをする
-v、--verbose	よりな
--no-diffs	リファクタリングのをしない
-w	されたファイルをきす
-n、--nobackups	されたファイルのバックアップをしない
-o OUTPUT_DIR、-- output-dir = OUTPUT_DIR	ファイルをきするのではなく、このディレクトリにファイルを きます。ファイルがされていない、バックアップ・ファイルはな ので、-nフラグがです。
-W、-きえなしのファイ ル	ファイルをきむでも、はありません。-oとに-oと、なソースツ リーがされ、コピーされます。インプラント-w
--add-suffix = ADD_SUFFIX	すべてのファイルにするをします。でないは-nがです。 --add- suffix='3' .py3 --add-suffix='3'は.py3ファイルをします。

2to3ツールは、Python 2.xでかかれたコードをPython 3.xコードにするためにされるPythonプログラムです。このツールはPython 2.xソースコードをみみ、のプログラムをしてなPython 3.xコードにします。

2to3ツールは、ほとんどすべてのコードをうなフィルターをむlib2to3としてライブラリでできます。lib2to3はライブラリなので、2to3ののフィルターをくことができます。

Examples

な

のPython2.xコードをえてみましょう。ファイルをexample.pyとしてし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)
```

のファイルには、いくつかののないがあります。raw_input()メソッドはPython 3.xではinput()にきえられ、printはもはやではなくです。このコードは、2to3ツールをしてPython 3.xコードにできます。

Unix

```
$ 2to3 example.py
```

Windows

```
> path/to/2to3.py example.py
```

のコードをすると、のようのソースファイルとのがされます。

```
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
```

は、`-w`フラグをしてソース・ファイルにきすことができます。 `-n`フラグがされていないり、のファイル `example.py.bak` バックアップがされ `example.py.bak` 。

Unix

```
$ 2to3 -w example.py
```

Windows

```
> path/to/2to3.py -w example.py
```

これで、 `example.py` ファイルがPython 2.xからPython 3.xコードにされました。

したら、 `example.py` はのなPython3.xコードをみます

Python 3.x 3.0

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

オンラインで2to3ツールをむ <https://riptutorial.com/ja/python/topic/5320/2to3> ツール

7: AMQPStormをしたRabbitMQの

バージョンのAMQPStormはpypiです。または、pipをしてインストールすることもできます

```
pip install amqpstorm
```

Examples

RabbitMQからメッセージをする

まず、ライブラリのインポートをします。

```
from amqpstorm import Connection
```

メッセージをするときは、にメッセージをするをするがあります。これはのびしなとすることができ、メッセージオブジェクト、またはメッセージタプル `to_tuple` された `to_tuple` パラメータにじてを `start_consuming` ん。

メッセージからデータをするに、メッセージをまたはするがあります。 RabbitMQにメッセージをしとしたことをらせるがあるので、これはです。

```
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)
```

に、RabbitMQサーバーへのをするがあります。

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

その、チャンネルをするがあります。はのチャンネルをつことができます。に、マルチスレッドのタスクをするは、スレッドごとに1つのスレッドをつことをおめしますではありません。

```
channel = connection.channel()
```

チャンネルをしたら、RabbitMQにメッセージのをさせておくがあります。これは、した `on_message` をして、すべてのメッセージをします。

RabbitMQサーバーで `simple_queue` ている `simple_queue` になります。また、RabbitMQでは、したすべてのメッセージがしたらそれをするようにしています。

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

に、RabbitMQサーバによってされたメッセージのをするためにIOループをするがあります。

```
channel.start_consuming(to_tuple=False)
```

RabbitMQにメッセージをする

まず、ライブラリのインポートをします。

```
from amqpstorm import Connection
from amqpstorm import Message
```

に、RabbitMQサーバーへのをくがあります。

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

その、チャンネルをするがあります。はのチャネルをつことができます。に、マルチスレッドのタスクをするは、スレッドごとに1つのスレッドをつことをおめしますではありません。

```
channel = connection.channel()
```

たちのチャンネルをしたら、メッセージをすることができます。

```
# Message Properties.
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
}

# Create the message.
message = Message.create(channel=channel, body='Hello World!', properties=properties)
```

これで、に `publish` をびして `routing_key publish` だけでメッセージをできます。この、メッセージを `simple_queue` というキューにします。

```
message.publish(routing_key='simple_queue')
```

RabbitMQでキューをする

まず、メイン・キューとキューの2つのチャネルをするがあります。のでは、なフラグをいくつか

していますが、コードのはしています。のような `confirm_delivery`、`delivery_mode` と `durable`。これらは、RabbitMQ [マニュアル](#) をしてください。

チャンネルをしたら、チャンネルからメインキューにメッセージをするためにできるメインチャンネルにバインディングをします。

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

に、れのメッセージをメインキューにするようにチャネルをするがあります。

```
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](#) メッセージ - *Time To Live*

これは、ののにキューのいメッセージをにするためにされますが、2つのオプションのをすることで、このをし、キューのメッセージのさをミリでできます。

- [x-dead-letter-routing-key](#)

このをすると、メッセージをにするデフォルトのではなく、れになったメッセージのをのキューにできます。

- [xデッドレター](#)

このは、メッセージを `hello_delay` から `hello` キューにするためにされたExchangeをします。

キューにする

すべてのなPikaパラメータをしたら、なパブリッシュをってキューにメッセージをするだけです。

```
delay_channel.basic.publish(exchange='',
                             routing_key='hello_delay',
                             body='test',
                             properties={'delivery_mod': 2})
```

スクリプトをすると、RabbitMQモジュールでのキューがされます。

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	

。

```

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")

```

オンラインでAMQPStormをしたRabbitMQのをむ

<https://riptutorial.com/ja/python/topic/3373/amqpstormをしたrabbitmqの>

8: ArcPy

ここでは、ArcPyのデータアクセスdaモジュールのカーソルをしています。

arcpy.da.SearchCursorのを、のよりいarcpy.SearchCursorとしないでください。

データアクセスモジュールarcpy.daは、ArcGIS 10.1でのみです。

Examples

カーソルをして、ファイルジオデータベースのフィーチャクラスのすべてののにして
1つのフィールドのをする

テンポラリフォルダC:\TempにあるテストファイルジオデータベースTest.gdbのテストフィーチャ
クラスTestFCからテストフィールドTestFieldをするには

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

createDissolvedGDB ワークスペースにファイルgdbをする

```
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
```

オンラインでArcPyをむ <https://riptutorial.com/ja/python/topic/4693/arcpy>

9: Asyncio モジュール

Examples

コルーチンとの

Python 3.5+ がリリースされるに、`asyncio` モジュールはジェネレータをしてびしをし、の Python 3.5 リリースとはなるをとっていました。

Python 3.x 3.5

Python 3.5 では、`async` キーワードと `await` キーワードがされました。 `await func()` びしの `await func()` にかっこがないことにしてください。

```
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

Python 3.5 よりには、`@asyncio.coroutine` デコレータをとってコルーチンをしていました。からのまりは、ジェネレータのにされました。 `yield from func()` まわりのかっこにしてください。

```
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

2つのをでするをすをにします。

```
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)
```

エグゼキュータ

Python 3.5のasync / awaitをします

asyncioは、タスクをにスケジュールするために`concurrent.futures`にある`Executor`オブジェクトのサポートしています。イベントループは、`Executor`オブジェクト、`Callable`、および`Callable`のパラメータをとる`run_in_executor()`をとっています。

のためのタスクのスケジュール`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))
```

イベントループはまた、「デフォルト」のつ`Executor`にりてなスロット`Executor`。 `Executor`をりて、ループからタスクをスケジュールするには、`set_default_executor()`メソッドをします。

```
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))

```

`concurrent.futures`には2つのタイプの`Executor`、`ThreadPoolExecutor`と`ProcessPoolExecutor`があります。`ThreadPoolExecutor`は、スレッドのプールがまれています。スレッドのプールは、コンストラクタをしてののスレッドにですることも、マシンのコアの5にすることもできます。`ThreadPoolExecutor`はスレッドプールをしてりてられたタスクをします。I/OバウンドではなくCPUバウンドでにれています。`ProcessPoolExecutor`とはに、りてられたタスクにしてしいプロセスをします。`ProcessPoolExecutor`は、`picklable`であるタスクとパラメータしかることができません。もな`picklable`タスクは、オブジェクトのメソッドです。`Executor`タスクとしてオブジェクトのメソッドをスケジュールするがあるは、`ThreadPoolExecutor`するがあります。

UVLoopの

`uvloop`は、`libuv`についた`asyncio.AbstractEventLoop`です`nodejs`によってされます。99の`asyncio`にしており、の`asyncio.EventLoop`よりもはるかに`asyncio.EventLoop`。、Windowsでは`uvloop`はできません
`pip install uvloop`。

```

import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Do your stuff here ...

```

つは、また、することで、イベントループのをすることができます`EventLoopPolicy` 1つに`uvloop`。

```

import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()

```

プリミティブイベント

のルーチンのスケジューリングをするには、`Event`をします。

にえば、イベントはっているレースでされたのようなものです。ランナーをブロックかられることができます。

```

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)

```

Bはっている
 はっている
 EVENT SET
 Bがトリガーされた
 コンシューマAがトリガされた

シンプルなWebSocket

ここでは、`asyncio` をつてなエコーwebsocketを`asyncio` ます。サーバーにしてメッセージをするためのコルーチンを行います。websocketのは、イベントループによってされる`main`コルーチンでされます。このは、[の](#)からされています。

```

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())

```

asyncioにするな

おそらくについてのものな `asyncio` のスレッドでにするジョブをするため、GILグローバルインタプリタロックをsidesteppingと-それはあなたがしてのタスクをできることです。それはしない

`asyncio` および `asyncio` とするためにされた `asyncio` は、コルーチンにされています。のでは `asyncio.sleep` にしてください。これは、バックグラウンドでっているブロッキングコルーチンであり、コールバック `await` びされたときにフローをします。 `time.sleep` は、ブロッキングのです。プログラムのフローはちょうどそこでし、 `time.sleep` に `time.sleep` ます。

のライブのは、ブロッキングのみではされている `requests` ライブラリです。 `asyncio` でそののいずれかをびすと、はわれません。 `aiohttp` ではでされた `asyncio` ので。コルーチンがにされます。

- してしたい、しているCPUバウンドタスクがあるは、 `asyncio` はあなたのためではありません。そのためには、 `threads` または `multiprocessing` がです。
- IOバウンドジョブをしているは、 `asyncio` をしてにできます。

オンラインで `Asyncio` モジュールをむ <https://riptutorial.com/ja/python/topic/1319/asyncio> モジュール

10: base64モジュール

き

Base64エンコーディングは、64をしてバイナリをASCIIにエンコードするなをします。base64モジュールはライブラリです。つまり、Pythonとにインストールされます。バイトとのはこのトピックにとってにであり、[ここですることが出来ます](#)。このトピックでは、Base64モジュールのさまざまなとベースをするについてします。

- base64.b64encodes、altchars = None
- base64.b64decodes、altchars = None、validate = False
- base64.standard_b64encodes
- base64.standard_b64decodes
- base64.urlsafe_b64encodes
- base64.urlsafe_b64decodes
- base64.b32encodes
- base64.b32decodes
- base64.b16encodes
- base64.b16decodes
- base64.a85encodeb、foldspaces = False、wrapcol = 0、pad = False、adobe = False
- base64.a85decodeb、foldspaces = False、adobe = False、ignorechars = b '\t\n\r\v'
- base64.b85encodeb、pad = False
- base64.b85decodeb

パラメーター

パラメータ	
base64.b64encode(s, altchars=None)	
s	バイトのようなオブジェクト
レター	Base64アルファベットをするときに、 '+'と '='をきえるさ2+ののバイトオブジェクト。なはされます。
base64.b64decode(s, altchars=None, validate=False)	
s	バイトのようなオブジェクト
レター	Base64アルファベットをするときに、 '+'と '='をきえるさ2+ののバイトオブジェクト。なはされます。

パラメータ	
	validateがTrueの、のBase64アルファベットまたはアルファベットにないは、パディングチェックにされません
base64.standard_b64encode(s)	
S	バイトのようなオブジェクト
base64.standard_b64decode(s)	
S	バイトのようなオブジェクト
base64.urlsafe_b64encode(s)	
S	バイトのようなオブジェクト
base64.urlsafe_b64decode(s)	
S	バイトのようなオブジェクト
b32encode(s)	
S	バイトのようなオブジェクト
b32decode(s)	
S	バイトのようなオブジェクト
base64.b16encode(s)	
S	バイトのようなオブジェクト
base64.b16decode(s)	
S	バイトのようなオブジェクト
base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)	
b	バイトのようなオブジェクト
foldspaces	foldspacesがTrueの、4つのしたスペースのわりに 'y' というがされます。
ラップコール	の0はをしません
パッド	padがTrueの、エンコードにバイトの4のにパディングされます
アドビ	adobeがTrueの、Adobeでされているように '<' と '>' でまれてエンコードされます

パラメータ	
<code>base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')</code>	
b	バイトのようなオブジェクト
foldspaces	<code>foldspaces</code> がTrueの、4つのしたスペースのわりに'y'というがされます。
アドビ	<code>adobe</code> がTrueの、Adobeでされているように'＜'と'＞'でまれてエンコードされます
ignorechars	エンコードするバイトのようなオブジェクト
<code>base64.b85encode(b, pad=False)</code>	
b	バイトのようなオブジェクト
パッド	<code>pad</code> がTrueの、エンコードにバイトの4のにパディングされます
<code>base64.b85decode(b)</code>	
b	バイトのようなオブジェクト

Python 3.4がするまでは、Base64のエンコーディングとデコードのは`bytes`または`bytes bytearray`でしか`bytearray`。、これらのはすべての**バイトオブジェクト**をくれます。

Examples

Base64のエンコードとデコード

スクリプトにbase64モジュールをめるには、まずそれをインポートするがあります

```
import base64
```

base64のエンコードとデコードのは、**バイトのようなオブジェクト**をとします。をバイトにするには、Pythonのみみエンコードをしてをエンコードするがあります。もには、`UTF-8`エンコーディングがされている、なるとをむこれらのなエンコーディングのしかしなリストをつけることができる**ここでは**のPythonドキュメントで。は、をバイトにエンコードするです。

```
s = "Hello World!"
b = s.encode("UTF-8")
```

ののはのようになります。

```
b'Hello World!'
```

`b`は、がバイトオブジェクトであることをすためにされます。

これらのバイトをBase64でエンコードするには、 `base64.b64encode()` をします。

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
print(e)
```

そのコードはをします

```
b'SGVsbG8gV29ybGQh'
```

これはまだバイトオブジェクトにあります。これらのバイトからをするには、Pythonの`decode()`メソッドをUTF-8エンコーディングでできます。

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
s1 = e.decode("UTF-8")
print(s1)
```

はのようになります。

```
SGVsbG8gV29ybGQh
```

をエンコードしてデコードするは、 `base64.b64decode()` メソッドをできます。

```
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)
```

どおり、はのになります。

```
Base64 Encoded: SGVsbG8gV29ybGQh
Hello World!
```

Base32のエンコードとデコード

base64モジュールには、Base32のエンコードおよびデコードも含まれています。これらは、Base64とよく使われます。

```
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)
```

これにより、のがされます。

```
Base32 Encoded: JBSWY3DPEBLW64TMMQ==
Hello World!
```

Base16のエンコードとデコード

base64モジュールには、Base16のエンコードおよびデコードも含まれています。ベース16は、もには**16**と呼ばれます。これらは、Base64とBase32のによく使われます。

```
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)
```

これにより、のがされます。

```
Base16 Encoded: 48656C6C6F20576F726C6421
Hello World!
```

ASCII85のエンコーディングとデコード

AdobeはBase85にしていますが、そのいがあり、そのエンコーディングとばれる**ASCII85**をしました。このエンコーディングは、Adobe PDFファイルでにされます。これらのはPython 3.4でリリースされました。そのの、`base64.a85encode()` および `base64.a85decode()` はのとです。

```
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)
```

これにより、がされます。

```
ASCII85 Encoded: 87cURD]i,"Ebo80
Hello World!
```

Base85のエンコードとデコード

Base64、Base32、Base16のとに、Base85のエンコーディングとデコードは`base64.b85encode()` と `base64.b85decode()` です。

```
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)
```

をします。

```
Base85 Encoded: NM&qnZy;Bla%^NF  
Hello World!
```

オンラインでbase64モジュールをむ <https://riptutorial.com/ja/python/topic/8678/base64モジュール>

11: ChemPy - pythonパッケージ

き

ChemPyは、に、のをしするためにされたpythonパッケージです。これは、のアプリケーションのオープンソースのPythonツールキットです。

Examples

の

```
from chempy import Substance
ferricyanide = Substance.from_formula('Fe(CN)63-')
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)63-, Fe(CN)<sub>6</sub><sup>3-</sup>
print('%3f' % ferricyanide.mass)
211.955
```

では、およびの0がキーとしてされ、のカウントがそれぞれのになりました。

のバランス

```
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%'}
```

バランスのとれた

```
from chempy import Equilibrium
from sympy import symbols
K1, K2, Kw = symbols('K1 K2 Kw')
e1 = Equilibrium({'MnO4-': 1, 'H+': 8, 'e-': 5}, {'Mn2+': 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
```

```
from chempy import Equilibrium
from chempy.chemistry import Species
water_autop = Equilibrium({'H2O': 1}, {'H+', 'OH-'}, 10**-14) # unit "molar" assumed
ammonia_prot = Equilibrium({'NH4+': 1}, {'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
```

イオン

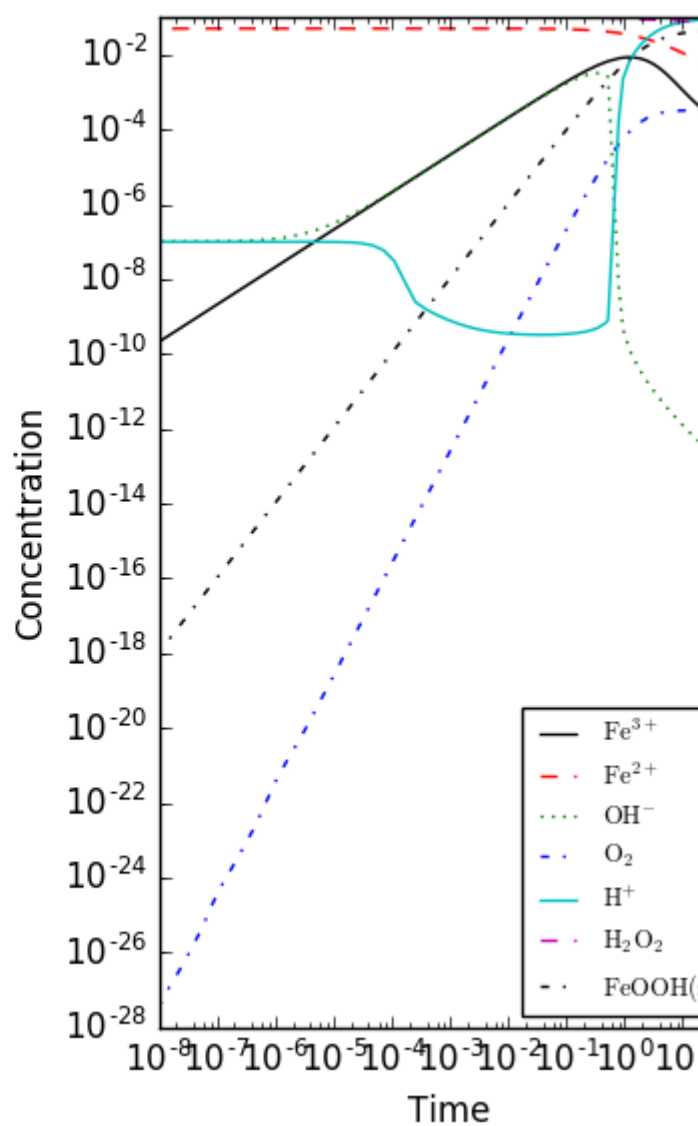
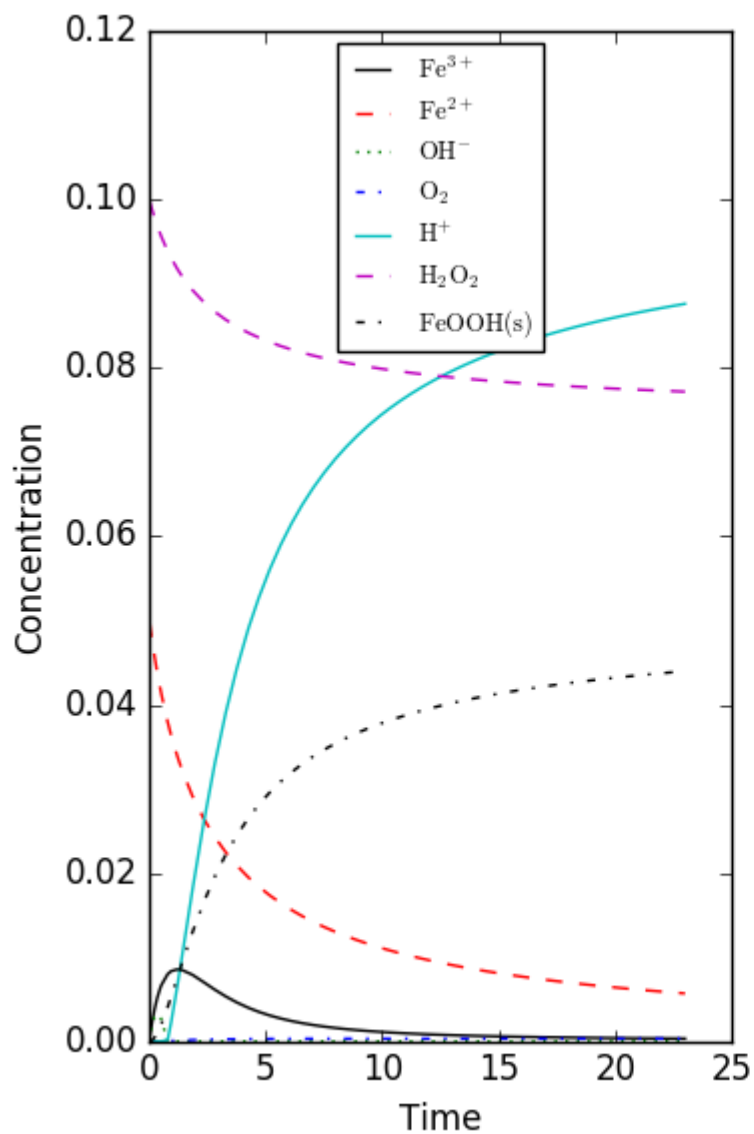
```
from chempy.electrolytes import ionic_strength
ionic_strength({'Fe+3': 0.050, 'ClO4-': 0.150}) == .3
True
```

```
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()

```



オンラインでChemPy - pythonパッケージをむ

<https://riptutorial.com/ja/python/topic/10625/chempy-----python/パッケージ>

12: configparser

き

このモジュールは、INIファイルになをするConfigParserクラスをします。これをして、エンドユーザがにカスタマイズできるPythonプログラムをくことができます。

- しいには、=でられたしいキーのペアがまれています
- キーはセクションでることができます
- INIファイルでは、セクションのタイトルは[]でまれてされています

ConfigParser.ConfigParser().getりはすべてです。evalおかげでよりなにできます

Examples

な

config.iniの

```
[DEFAULT]
debug = True
name = Test
password = password

[FILES]
path = /path/to/file
```

Pythonで

```
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'
```

プログラムによるファイルの

ファイルにはセクションがあり、セクションにはキーとがまれています。configparserモジュールは、ファイルを読み取るためにできます。ファイルの -

```
import configparser
config = configparser.ConfigParser()
config['settings']={'resolution':'320x240',
                   'color':'blue'}
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

ファイルにはのがまれています

```
[settings]
resolution = 320x240
color = blue
```

のフィールドをするは、フィールドをしてをりてます

```
settings=config['settings']
settings['color']='red'
```

オンラインでconfigparserをむ <https://riptutorial.com/ja/python/topic/9186/configparser>

13: CSVのみき

Examples

TSVファイルの

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'])
```

ファイル

```
$ cat /tmp/output.tsv

name      field
Dijkstra   Computer Science
Shelah     Math
Aumann     Economic Sciences
```

パンダをう

dictまたはDataFrameからCSVファイルをします。

```
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")
```

CSVファイルをDataFrameとしてみみ、それをdict

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

オンラインでCSVのみきをむ <https://riptutorial.com/ja/python/topic/2116/csvのみき>

14: ctypes

き

`ctypes`は、コンパイルされたネイティブライブラリからエクスポートされたをびすPythonビルトインライブラリです。

このライブラリはコンパイルされたコードをするため、OSにします。

Examples

な

`libc`の`ntohl`をいたいとしましょう。

まず、`libc.so`をロード`libc.so`があります。

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

に、オブジェクトをします。

```
>>> ntohl = libc.ntohl
>>> ntohl
<_FuncPtr object at 0xbaadf00d>
```

そして、にをびすことができます

```
>>> ntohl(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

たちがしていることとまったくじです。

のとし

ファイルのみみにしました

にえられるエラーは、ライブラリをロードできません。その、は`OSError`がします。

これは、ファイルがないか、またはOSによってされないためです。

```
>>> 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
```

あなたがることができるように、エラーはらかであり、かなりです。

2のは、ファイルがつかりましたが、しいではありません。

```
>>> 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
```

この、ファイルはスクリプトファイルであり、`.so`ファイルではありません。これは、Linuxマシンで`.dll`ファイルをくときや、32ビットのPythonインタプリタで64ビットファイルをくときにもこります。このように、この、エラーはもうしで、いくらかりげるがあります。

へのアクセスにしました

`.so`ファイルをにみんだとすると、ののようになにアクセスするがあります。

しないがされると、`AttributeError`がします。

```
>>> 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
```

ctypes オブジェクト

もなオブジェクトは`int`です

```
>>> obj = ctypes.c_int(12)
>>> obj
c_long(12)
```

、`obj`は12をむメモリのチャンクをします。

それにアクセスしたり、したりすることもできます。

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

`obj`はメモリのをすので、サイズともべることができます

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

ctypes

れたCプログラマがっているように、のはあなたをそれほどざけません。にたちがくのはです

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

これはのではありませんが、かなりいですたちは、16 `int`のをすクラスをしました。

これですだけです。

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcafe>
```

、`arr`は0から15までのをむのです。

らはどんなリストとにアクセスすることができます

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

また、の`ctypes`オブジェクトとに、サイズともっています

```
>>> sizeof(arr)
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc00010ff'
```

ctypesのラッピング

によっては、Cはポインタをけります。な`ctypes`ユーザーとして、たちはこれらのをしたいし、

Pythonをとしてすことさえしたい。

をしましょう

```
>>> def max(x, y):
    return x if x >= y else y
```

これは2つのをとり、じのをします。こののために、typeがintであるとしましょう。

のとに、そのプロトタイプをすオブジェクトをすることができます

```
>>> CFUNCTYPE(c_int, c_int, c_int)
<CFunctionType object at 0xdeadbeef>
```

このプロトタイプは、c_int のをすをし、2つのc_intのをc_intます。

はをラップしましょう

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

プロトタイプには、さらにくのがあります。は libc.ntohlのように ctypesをラップし、をびすときにしいがされていることをできます。

```
>>> 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)
```

な

のすべてのを1つのなシナリオにしましょう libcのlfindをう。

こののについては、[マニュアルページをしてください](#)。はあなたにそれをけるにそれをむことをくめます。

まず、なプロトタイプをします。

```
>>> 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)
```

に、をしましょう

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

に、をします。

```
>>> def compar(x, y):  
    return x.contents.value - y.contents.value
```

`x`と`y`は`POINTER(c_int)`、にメモリにされているをするには、それらをしてをるがあることにしてください。

これですべてをみわたせることができます

```
>>> lfind = lfind_proto(libc.lfind)  
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```

`ptr`はされた`void`ポインタです。 `arr key`がつからなかった、は`None`になりますが、このはながられます。

これでしてにアクセスできます

```
>>> cast(ptr, POINTER(c_int)).contents  
c_long(12)
```

また、`ptr`が`arr`のしいをしていることがかります。

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr  
True
```

オンラインでctypesをむ <https://riptutorial.com/ja/python/topic/9050/ctypes>

15: Dequeモジュール

- `dq = deque`のキューをします。
- `dq = dequeiterable`いくつかのをつキューをします。
- `dq.appendobject`オブジェクトをキューのにする
- `dq.appendleftobject`オブジェクトをキューのにします。
- `dq.pop` - `> object`ものオブジェクトをしてします
- `dq.popleft` - `> object`のオブジェクトをしてします
- `dq.extenditerable`キューのにいくつかのをします
- `dq.extendleftiterable`いくつかのをキューのにします

パラメーター

パラメータ	
<code>iterable</code>	のオブジェクトからコピーされたをつキューをします。
<code>maxlen</code>	キューのきさをし、しいがされたときにいをしします。

このクラスは、いずれかのからのおよびポップをにする [リスト](#) にたオブジェクトがなにですの `deque` は「ダブルエンドキュー」をします。

されるメソッドはにはにてい `append`、`pop`、`append`、または `extend` ようなものには `left` にをけることができます。01でうことができるので、のをにおよびするがあるは、`deque` データをリストよりするがあります。

Examples

している `deque`

このクラスでななメソッドは、`popleft` と `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])
```

デュークサイズをする

キューをするに `maxlen` パラメータをして、キューのサイズをする

```
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)
```

キューでなメソッド

のキューをする

```
dl = deque() # deque([]) creating empty deque
```

いくつかのdequeをする

```
dl = deque([1, 2, 3, 4]) # deque([1, 2, 3, 4])
```

dequeにをする

```
dl.append(5) # deque([1, 2, 3, 4, 5])
```

デキューののをする

```
dl.appendleft(0) # deque([0, 1, 2, 3, 4, 5])
```

dequeにのリストをする

```
dl.extend([6, 7]) # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

からのリストをする

```
dl.extendleft([-2, -1]) # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

`.pop()` をすると、にからがされます。

```
dl.pop() # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

`.popleft()` をしてアイテムをからする

```
dl.popleft() # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

そのでをする

```
dl.remove(1) # deque([-2, 0, 2, 3, 4, 5, 6])
```

キューののをする

```
dl.reverse() # deque([6, 5, 4, 3, 2, 0, -2])
```

い

Dequeは、のキューをつのPythonデータです。 `queue.Queue`、それがスレッドのためのものなので、はしていない。キューのなユースケースは、。

```
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
```

なグラフがあるとします

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

からのをつけることができます

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

オンラインでDequeモジュールをむ <https://riptutorial.com/ja/python/topic/1976/deque>モジュール

16: Functools モジュール

Examples

な

`partial` は、のからアプリケーションをします。これは、のまたはキーワードのいくつかにをバインドし、すでにされているをたないびしオブジェクトをするためにされます。

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('callable')
3390155550
```

`partial()` は、がするように、のなをにします。のをてみましょう

```
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
```

`g` がされると、4つの `a, b, c, x` をる `f` もの3つの `a, b, c`, `f` は `g` がびされるとし、`g(2)` は `f` 4のをします。

`partial` をえるの1つはシフトレジスタです。の1つのをいくつかのにしんだ。`partial` は、データがストリームとしてされ、のをすことができないにです。

total_ordering

`__eq__()` なクラスをするには、メソッド `__eq__()`、`__lt__()`、`__le__()`、`__gt__()` および `__ge__()` をするがあります。

クラスにされた `total_ordering` デコレータは、`__eq__()` `__lt__()`、`__le__()`、`__gt__()` および `__ge__()` で `__eq__()` と1 `__eq__()` のをし、クラスのすべてののけをします。

```
@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))

```

デコレータは、されたメソッドとのコンポジションをして、のメソッドをします。たとえば、`__lt__()`と`__eq__()`をし、`__lt__()`をさせたい`__gt__()`では`not __lt__()` and `not __eq__()`にチェックできます。

`total_ordering`は、Python 2.7でのみできます。

らす

Pythonの3.xのでは、`reduce`にした[ここでは](#)、みみからされており、からしなければならない `functools`。

```

from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))

```

lru_cache

`@lru_cache`デコレータは、[もされていない](#)キャッシュをして、でのをラップするためにできます。これにより、びしをメモすることができ、じパラメータをしたのびしをしなくてもにすことができます。

```

@lru_cache(maxsize=None) # Boundless cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)

```

のでは、の`fibonacci(3)`あれば、だけ、される`fibonacci` LRUキャッシュをっていないかった、`fibonacci(3)` 230のにされたであろう。したがって、`@lru_cache`はやプログラミングにに `@lru_cache`ます。なをしなパラメータでびすことができます。

`@lru_cache`は2つのがあります

- `maxsize` するコールの。ユニークなコールのが`maxsize`をえると、LRUキャッシュはもにされたコールをします。
- `typed` 3.3でなるののがなるキャッシュレコードにしているかどうかをするためのフラグですつまり、`3.0`と`3`がなるとしてカウントされる

キャッシュのもされます。

```

>>> fib.cache_info()

```

```
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

`@lru_cache`はをしてをキャッシュするので、のすべてのパラメータは、キャッシュがするためにハッシュでなければなりません。

`@lru_cache`[Pythonドキュメント](#)。 `@lru_cache`は3.2でされました。

`cmp_to_key`

Pythonは、キーをけるメソッドのソートをしました。これらのはをとり、をソートするためにされるキーをします。

いは、2つのをるためにされ、のが2のとじかさいは-1、0または+1をします。これは、しいキーとがありません。

それは`functools.cmp_to_key`がる`functools.cmp_to_key`

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

[Pythonライブラリのドキュメント](#)からられた

オンラインで[Functoolsモジュール](#)をむ <https://riptutorial.com/ja/python/topic/2492/functools>モジュール

17: groupby

き

Pythonでは、`itertools.groupby()` メソッドをすると、は、されたプロパティについてなクラスのものをのセットにグループできます。

- `itertools.groupbyiterable`、`key = None`またはsome function

パラメーター

パラメータ	
りしな	のpython iterable
キー	イテラブルをグループする

`groupby`はいにくいですが、にすべきなルールはのとおりです。

グループにするキーとじキーをして、グループするアイテムをにべえます

がここのをて、クラスをとてどのようにされているかをするをおめします。

Examples

1

あなたはをっているとう

```
s = 'AAAABBBCCDAABBB'
```

あなたはそれをして、すべての'A'が1つのリストにっているので、すべての'B'と'C'などで、このようなことをすることができます

```
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
```

は

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'],
 'B': ['B', 'B', 'B', 'B', 'B', 'B'],
 'C': ['C', 'C'],
 'D': ['D']}
```

しかし、きなデータセットの、これらのアイテムをメモリにしています。これは、`groupby`が
です

のようにすることで、よりなでじをることができました

```
# 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
```

は

```
{'A': ['A', 'A'], 'B': ['B', 'B', 'B'], 'C': ['C', 'C'], 'D': ['D']}
```

`group by`をしたときの「A」のは、の「A」ののよりもないことにしてください。のように、
`s`のをソートしてから`c`にすことでがわれるのをぐことができます

```
c = groupby(sorted(s))

dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

は

```
{'A': ['A', 'A', 'A', 'A', 'A', 'A'], 'B': ['B', 'B', 'B', 'B', 'B', 'B'], 'C': ['C', 'C'],
 'D': ['D']}
```

たちはすべてのAをとっています。

2

このでは、もしなかったのデフォルトのキーのをします。

```
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
```

は

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
10: [10],
11: [11],
 'goat': ['goat']}
```

タプルがこのリストの1つのキーとしてカウントされることにしてください

3

ここでは、mulatoとcamelがにされないことにしてください。されたキーをつののみがされます。cのは、2つののをにします。しかし、じバージョンのキーでにソートされたデータがあるしいバージョンをてください。

```
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
```

は

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

ソートみバージョン

```
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
```

は

```
['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']}
```

4

ここでは、なるタイプのりしをするときにがこるかをしています。

```
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
```

は

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
 'plant': [('plant', 'cactus')],
 'vehicle': [('vehicle', 'harley'),
             ('vehicle', 'speed boat'),
             ('vehicle', 'school bus')]}
```

のは、にはそれとじです。のいは、すべてのタプルをリストにしたことです。

```
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
```

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
 'plant': [['plant', 'cactus']],
 'vehicle': [['vehicle', 'harley'],
             ['vehicle', 'speed boat'],
             ['vehicle', 'school bus']]}
```

オンラインでgroupbyをむ <https://riptutorial.com/ja/python/topic/8690/groupby-->

18: GZipでめる

き

このモジュールは、GNUプログラムgzipやgunzipのように、ファイルをしするなインタフェースをします。

データは、zlibモジュールによってされます。

gzipモジュールは、PythonのファイルオブジェクトのにモデルされたGzipFileクラスをします。GzipFileクラスはgzipのファイルのみし、データをにまたはしてのファイルオブジェクトのようになします。

Examples

GNU zipファイルのみき

```
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)
```

ターミナルから1gzip_write.py1.Runとしてしてください。

```
$ python gzip_write.py

application/x-gzip; charset=binary
example.txt.gz contains 68 bytes of compressed data
```

オンラインでGZipでめるをむ <https://riptutorial.com/ja/python/topic/8993/gzipでめる>

19: HTML

Examples

BeautifulSoupののにテキストをする

のHTMLがあるとします。

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

labelののにテキスト「John Smith」をするがあります。

この、labelをテキストでし、.next_siblingプロパティをすることが.next_siblingます。

```
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())
```

John Smithします。

BeautifulSoupでCSSセクタをう

BeautifulSoupはCSSセクタのサポートがされていますが、もにされています。select()メソッドをしてのをし、select_one()をしてのをします。

な

```
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())
```

```
item1
item2
item3
```

PyQuery

pyqueryはPythonのjqueryライクなライブラリです。それはCSSセレクタをによくサポートしています。

```
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)
```

オンラインでHTMLをむ <https://riptutorial.com/ja/python/topic/1384/html>

20: ijson

き

ijsonはPythonでJSONファイルをうらしいライブラリです。ながら、デフォルトではなPython JSONパーサーをバックエンドとしてしています。Cのバックエンドをすると、はるかにいパフォーマンスをできます。

Examples

な

サンプルの1つのベンチマークから

```
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ファイル [リンク](#)

オンラインでijsonをむ <https://riptutorial.com/ja/python/topic/8342/ijson>

21: Itertoolsモジュール

- `import itertools`

Examples

をしてオブジェクトからをグループする

グループするがあるりしなものからする

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

グループされたジェネレータをし、タブルの2のでグループします。

```
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)]
```

するのグループのみがグループされます。 `groupby for Eg`をびすにじキーでソートするがあるかもしれませんのがされます

```
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)]
```

`groupby`によってされたグループは、のりしのになるイテレータです。たとえば、グループをキーでソートするは、のようにはしません。グループ2がフェッチされると、グループ5はになります

```
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 []
```

ソートをしくうには、ソートにイテレータからリストをします

```
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)]
```

のスライスをする

Itertools "islice"では、ジェネレーターをスライスすることができます

```
results = fetch_paged_results() # returns a generator
limit = 20 # Only want the first 20 results
for data in itertools.islice(results, limit):
    print(data)
```

、ジェネレータをスライスすることはできません

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[3:]:
    print(part)
```

あげる

```
Traceback (most recent call last):
  File "gen.py", line 6, in <module>
    for part in gen()[3:]:
TypeError: 'generator' object is not subscriptable
```

しかし、これはします

```
import itertools

def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in itertools.islice(gen(), 3):
    print(part)
```

のスライスとに、 start 、 stop 、 step もできます。

```
itertools.islice(iterable, 1, 30, 3)
```

itertools.product

これは、りしなリストのデカルトをすることをにします。

えは、

```
for x, y in itertools.product(xrange(10), xrange(10)):
    print x, y
```

は

```
for x in xrange(10):
    for y in xrange(10):
        print x, y
```

のをけるすべてのPythonとに、*をして、するためにitertools.productにリストをすことができます。

このように、

```
its = [xrange(10)] * 2
for x,y in itertools.product(*its):
    print x, y
```

ののとじがられます。

```
>>> 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

き

このなは、のをします。えは...

```
for number in itertools.count():
```

```
if number > 20:
    break
print(number)
```

々はれなければならないこと、あるいははにすることにしてください

```
0
1
2
3
4
5
6
7
8
9
10
```

`count()` は `start` と `step` 2つのをります

```
for number in itertools.count(start=10, step=4):
    print(number)
    if number > 20:
        break
```

```
10
14
18
22
```

itertools takewhile

`itertools.takewhile` をすると、シーケンスのをにが `False` なるまでできます。

```
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)
```

これは `[0, 2, 4, 12, 18]` `0`、`2`、`4`、`12`、`18` `[0, 2, 4, 12, 18]` します。

にするのすなわち、ブールをす `is_even` は、`13` ことにしてください。 `takewhile` がえられたにして `False` をするにつたら、それはれます。

される `takewhile` のコードからされたにしています。

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

`takewhile`と`dropwhile`によってされたをすると、の`iterable`がされます。

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

itertools.dropwhile

`itertools.dropwhile`をすると、がに`False`にシーケンスからアイテムをできます。

```
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)
```

これは `[13, 14, 22, 23, 44]` `13,14,22,23,44` `[13, 14, 22, 23, 44]` します。

このは、`takewhile`と同じ`takewhile`が、`dropwhile`をして`dropwhile`ます。

にするのすなわち、ブールをす `is_even`は、`13`ことにしてください。そののすべてののはされます。

される `dropwhile`のコードからされたにしています。

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

`takewhile`と`dropwhile`によってされたのはの`iterable`をします。

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

のイテレーターがいたされるまで2つのイテレーターをする

みみ`zip()`とに、`itertools.zip_longest`は2つの`itertools.zip_longest`うちのいのをえてをけます。

```
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)
```

オプションの `fillvalue` をすことができますデフォルトは `..`。

```
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)
```

Python 2.6と2.7では、これは `itertools.izip_longest` とばれてい `itertools.izip_longest`。

itertoolsモジュールのみわせ

`itertools.combinations` は、リストの k -コンビネーションシーケンスのジェネレータをします。

いえれば、リストのすべての k -wise みわせたタプルのジェネレータをします。

えは

リストがある

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 2))
print b
```

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

のは、リスト `a` のすべての k -wise みわせたタプルのリストにされたで `a`

3つのすべてのみわせたをつけることもできます

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 3))
print b
```

```
[(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)]
```

のイテレータをさせる

`itertools.chain` をして、のジェネレータのを `itertools.chain` にするのジェネレータをします。

```
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))
```

```
'1 2 3 4 x y z'
```

のコンストラクタとして、クラスメソッド `chain.from_iterable` をできます。このメソッドは、のパラメータとして `iterable` のりしなものをとります。とじをするには

```
' '.join(chain.from_iterable([a,b]))
```

`chain` はののをすることができますが、 `chain.from_iterable` はの `iterable` をするのです。

itertools.repeat

n 回繰り返す

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...     print(i)
over-and-over
over-and-over
over-and-over
```

りしなのを繰り返す

Python 3.x 3.2

`accumulate` と、のまたは `accumulate` られます。

```
>>> 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]
```

イテレータのをさせる

`cycle` はのです。

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

したがって、ループをけるためにこれをするは、をえるようにしてください。

```
>>> # 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` は、`r` の長さのジェネレータを返します。

```
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)]
```

リストの長さ `r` を指定して、その長さのジェネレータを返します。返される `set` はユニークなものを返します。

```
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)}
```

オンラインで `itertools` モジュールをみる <https://riptutorial.com/ja/python/topic/1564/itertools> モジュール

22: JSONモジュール

バージョンのをむなドキュメントについては、[ドキュメント](#)をしてください。

タイプ

デフォルト

`json`モジュールはデフォルトでのエンコーディングとデコードをします

デシリアライゼーションタイプ

JSON	Python
オブジェクト	dict
アレイ	リスト
	str
int	int
	<
ヌル	し

`json`モジュールは、するfloatとして`NaN`、`Infinity`、および`-Infinity`もしています。これはJSONのです。

シリアライゼーションタイプ

Python	JSON
dict	オブジェクト
リスト、タプル	アレイ
str	
int、float、int / floatからされたEnums	

Python	JSON
し	ヌル

NaN、Infinity、および-Infinity エンコードをしないには、`allow_nan=False` エンコードするがあります。これらのをエンコードしようとすると、`ValueError` ます。

カスタムデシリアル

なったがなデータをうためのさまざまなフックがあります。`functools.partial` すると、これらのにするパラメータをにすることができます。

シリアライゼーション

オブジェクトをするにオブジェクトでするをすることができます。

```
# 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)
```

デシリアライゼーション

jsonによってされる、`object_hook`や`parse_float`など、さまざまなフックがあります。おいのPythonのバージョンをしたリストは、[こちらをご覧ください](#)。

```
# 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)
```

さらにカスタムデシリアライズ

json モジュールでは、 `json.JSONEncoder` と `json.JSONDecoder` を/してそののをすることもできます。のフックは、ののメソッドをすることでデフォルトとしてすることができます。これらをするには、クラスを `cls` パラメータとしてするにします。 `functools.partial` すると、これらのに `cls` パラメータをにすることができます。

```
# 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

Python dictからJSONをする

```
import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)
```

のスニペットは、をします

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

JSONからPython dictをする

```
import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)
```

のスニペットは、をします

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

データをファイルにする

のスニペットは、`d`にされたデータをJSONにエンコードし、ファイルにします `filename` をの `filename` できえます。

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

ファイルからのデータの

のスニペットは、JSONでエンコードされたファイルをき `filename` をの `filename` できえます、ファイルにされているオブジェクトをします。

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

‘load’と‘dump’と‘dump’と‘dump’

`json` モジュールには、Unicodeのみるときみ、およびファイルのみきをうためのがまれています。これらは、のに`s`でされます。これらのでは、`StringIO`オブジェクトをしています、じがファイルのようなオブジェクトにもされます。

ここでは、ベースのをします。

```
import json

data = {"foo": "bar", "baz": []}
json_string = json.dumps(data)
# u'{"foo": "bar", "baz": []}'
json.loads(json_string)
# {"foo": "bar", "baz": []}
```

ここではファイルベースのをします

```
import json

from io import StringIO

json_file = StringIO()
data = {"foo": "bar", "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": []}

```

ないがわかるように、jsonデータをダンプするときは、りをりむのではなく、ファイルハンドルをにすがあります。また、データのをけるために、みきのにファイルのにをけるがあることにもしてください。ファイルをくとき、カーソルは₀にかれるので、もです

```

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": []}

```

jsonデータをうのをつことで、pysparkのjson-per-lineのようにjsonをpysparkしたフォーマットでかつにすることができます。

```

# 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')

```

コマンドラインから**JSON**をきれいにするために`json.tool`をびします

のようなJSONファイル "foo.json"があるとします。

```

{"foo": {"bar": {"baz": 1}}}

```

モジュールをコマンドラインからびすことができますファイルをととしてして、それをきれいにすることができます

```

$ python -m json.tool foo.json
{
  "foo": {
    "bar": {
      "baz": 1
    }
  }
}

```

モジュールはSTDOUTからのものになりますので、Bashではじことができます

```
$ cat foo.json | python -m json.tool
```

JSONの

のデータがあるとしています。

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

これをJSONとしてダンプするだけで、なことはありません

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

よりきれいなをるためにインデントをする

きれいなをしたいは、`indent` サイズをすることができます

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```

したをるためにアルファベットにキーをべえる

デフォルトでは、のキーのはです。たちはいつもじをるためにアルファベットにそれらをることができます

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

コンパクトなをるためにをりく

デフォルトの`,`と`:`とはなるセパレータをすることで、なスペースをりきたいかもしれませ

ん

```
>>>print(json.dumps(data, separators=(',', ':')))  
{"cats":[{"name":"Tubbs","color":"white"}, {"name":"Pepper","color":"black"}]}
```

カスタムオブジェクトをエンコードするJSON

たちがのこをしてみたら

```
import json  
from datetime import datetime  
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}  
print(json.dumps(data))
```

TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializableというエラーが
TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializableます。

datetimeオブジェクトをにシリアルできるようにするには、のカスタムコードをするがあります
。

```
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)
```

json.dumpsわりにこのエンコーダクラスをします。

```
encoder = DatetimeJSONEncoder()  
print(encoder.encode(data))  
# prints {"datetime": "2016-09-26T04:44:00"}
```

オンラインでJSONモジュールをむ <https://riptutorial.com/ja/python/topic/272/json>モジュール

23: kivy - NUIのためのクロスプラットフォーム Python フレームワーク

き

NUIユーザインタフェースNUIは、とコンピュータとののためのシステムであり、ななのにするなをじてユーザがする。

Kivyは、なるデバイスにインストールできるマルチタッチのメディアリッチアプリケーションのPythonライブラリです。マルチタッチは、2つのからのをにまたはするためのタッチ、タッチスクリーンまたはトラックパッドのをす。

Examples

のアプリ

Kivyアプリケーションをするには

1. サブクラス**app**クラス
2. ビルドメソッドをすると、ウィジェットがされます。
3. をひすクラスをインスタンスします。

```
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()
```

```
from kivy.app import App
```

のステートメントはクラス**app**をインポートします。これは、インストールディレクトリの `your_installtion_directory / kivy / app.py`にあります

```
from kivy.uix.label import Label
```

のステートメントは、**uxLabel**をインポートします。すべてのuxは、インストールディレクトリ `your_installation_directory / kivy / uix /`にあります。

```
class Test(App):
```


のステートメントはあなたのアプリケーションをするためのもので、クラスはあなたのアプリになります。このクラスは、のappクラスをしています。

```
def build(self):
```

のステートメントは、appクラスのビルドメソッドをオーバーライドします。あなたがアプリをするときにするがあるウィジェットをします。

```
return Label(text='Hello world')
```

のステートメントは、ビルドメソッドのです。これは、 **Hello World**のテキストでLabelをしています。

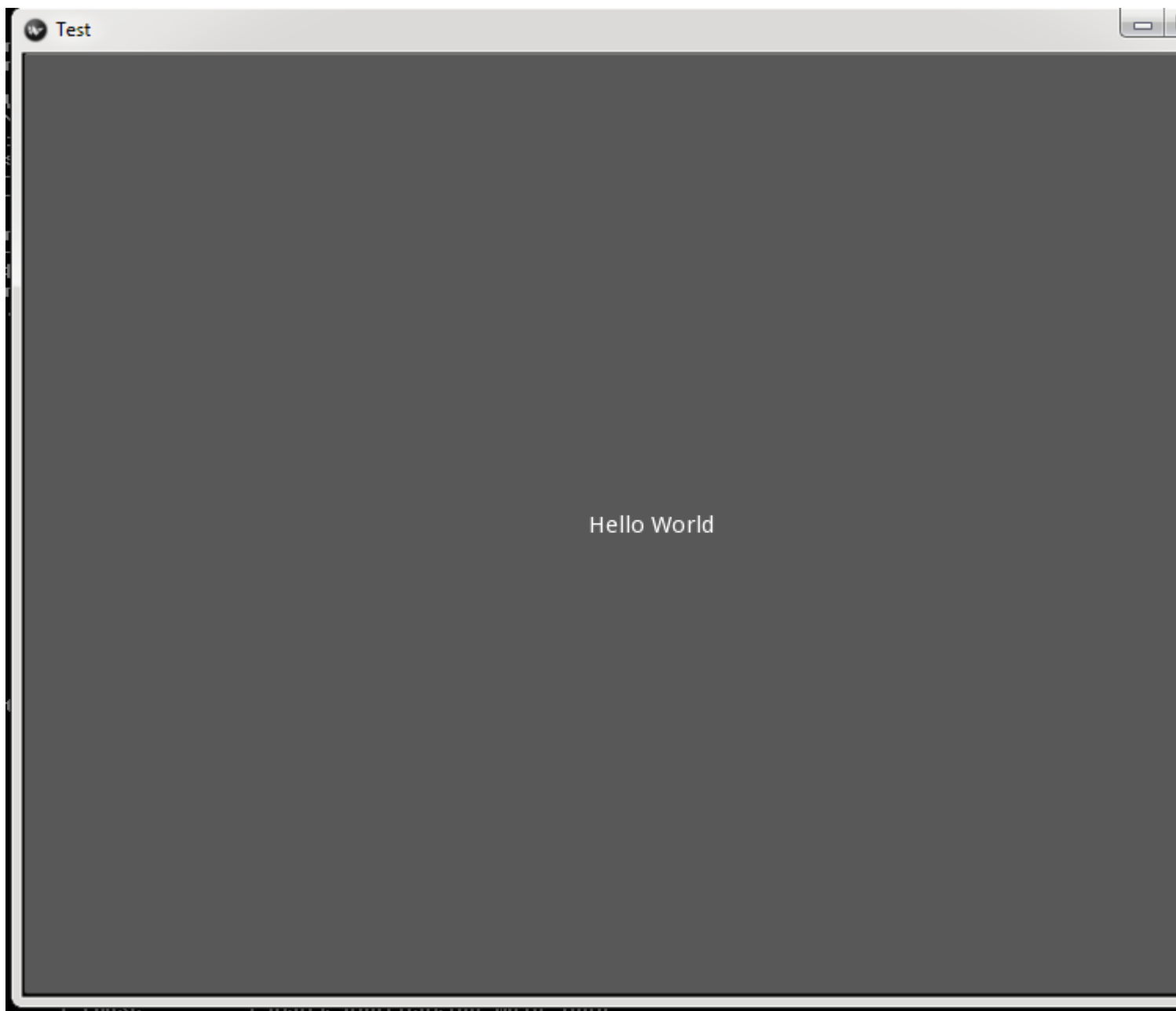
```
if __name__ == '__main__':
```

のステートメントは、Pythonインタプリタがあなたのアプリのをするエントリポイントです。

```
Test().run()
```

のステートメントは、そのインスタンスをしてTestクラスをします。そして、appクラスrunをびします。

あなたのアプリはののようになります。



オンラインでkivy - NUIのためのクロスプラットフォームPythonフレームワークをむ

<https://riptutorial.com/ja/python/topic/10743/kivy-----nuiのためのクロスプラットフォームpythonフレームワーク>

24: Matplotlibでプロットする

き

Matplotlib <https://matplotlib.org/>は、NumPyに基づく2Dプロットのためのライブラリです。いくつかのなががあります。よりくのは、<https://matplotlib.org/2.0.2/gallery.html>および<https://matplotlib.org/2.0.2/examples/index.html>および[http //www.riptutorial.com/topic/881](http://www.riptutorial.com/topic/881)

Examples

Matplotlibのなプロット

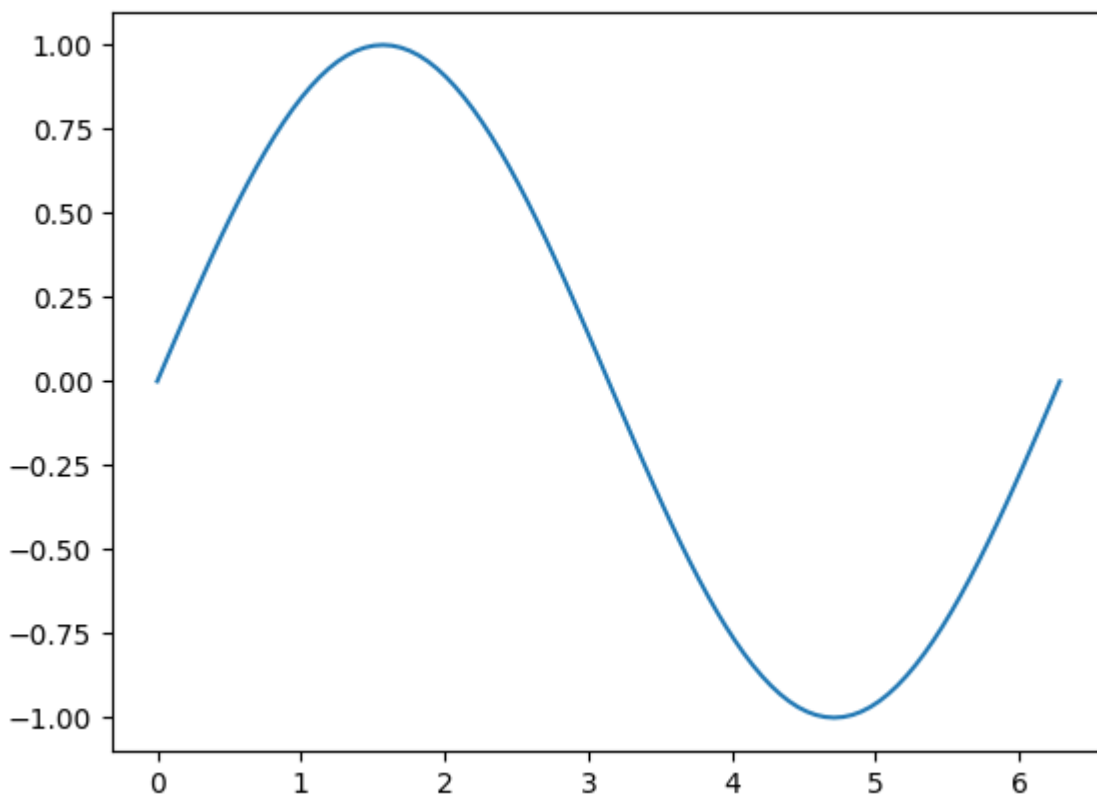
このは、**Matplotlib**をしてなサインカーブをするをしています

```
# 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()
```



なプロットにくのフィーチャをするのラベル、タイトル、のり、グリッド、

このでは、サインカーブプロットをし、それにくのフィーチャをします。タイトル、のラベル、タイトル、のり、グリッド、がされます。

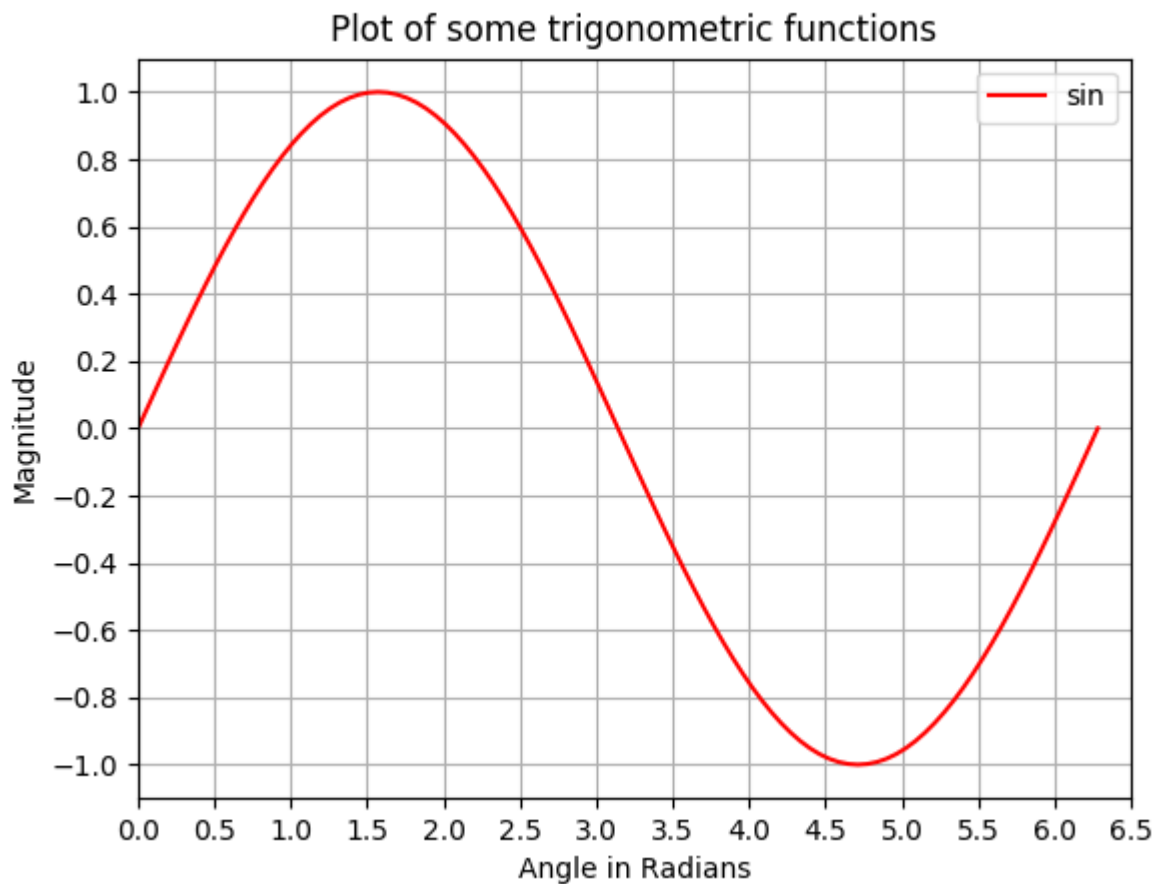
```
# Plotting tutorials in Python
# Enhancing a plot

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()
```



MATLABにたスーパーインポーズによってじにのプロットをる

このでは、プロットをねわせることにより、サインカーブとコサインカーブがじにプロットされています。

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using single plot command and legend

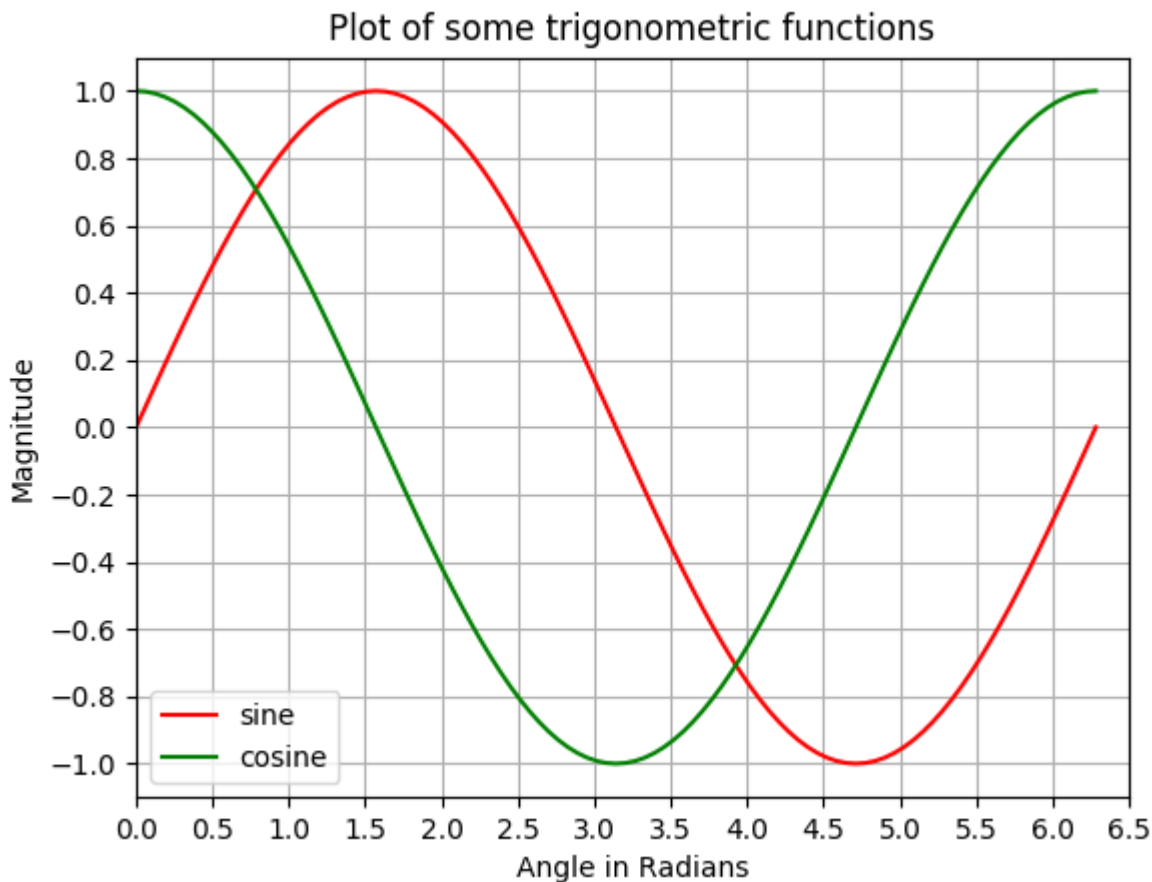
import numpy as np
import matplotlib.pyplot as plt

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()
```



々のプロットコマンドをしてプロットのねわせをしてじにのプロットをする

のとに、ここでは、々のプロットコマンドをして、じにとをプロットします。これはもっと Pythonicであり、プロットにして々のハンドルをるためにうことができます。

```
# 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

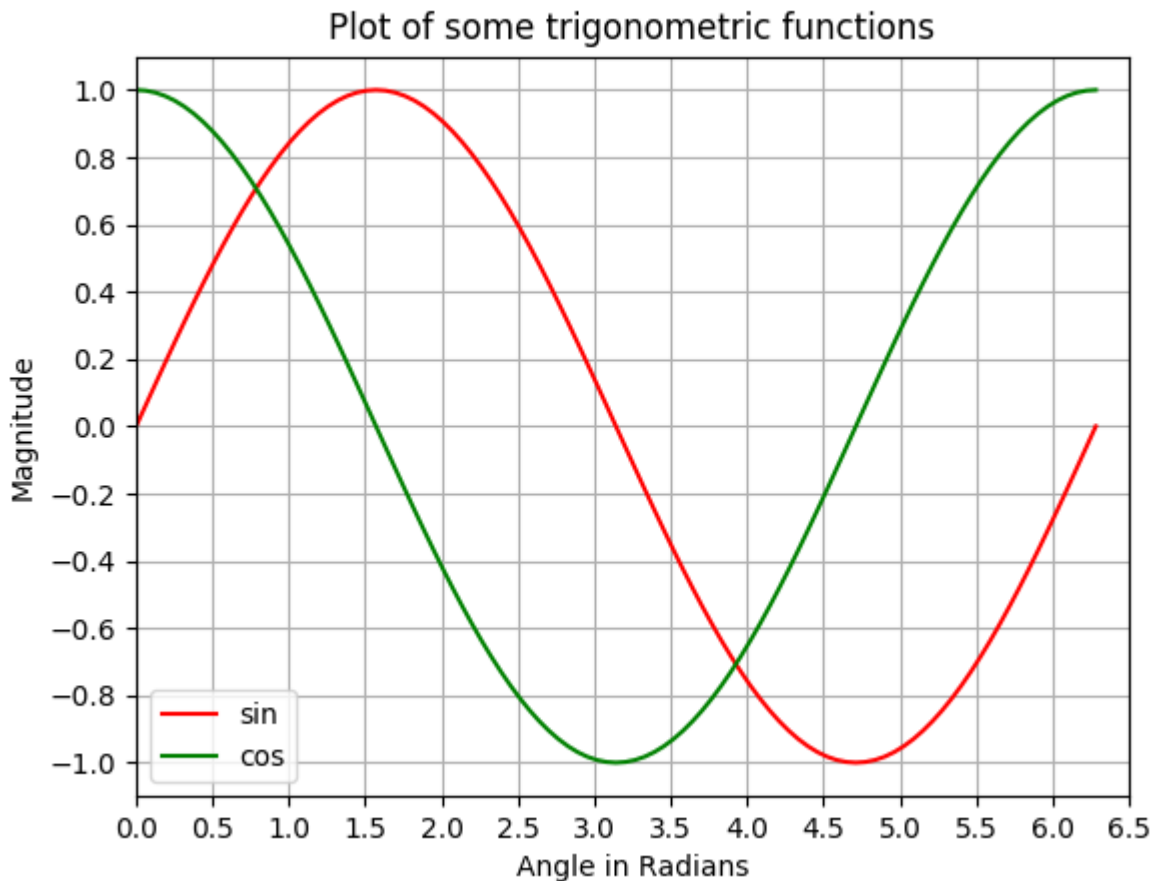
import numpy as np
import matplotlib.pyplot as plt

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()
```



XだがYがなるプロット **twinx**をう

このでは、**y**がなるの**x**をつじプロットのとをプロットします。これは、**twinx**コマンドをしています。

```
# 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

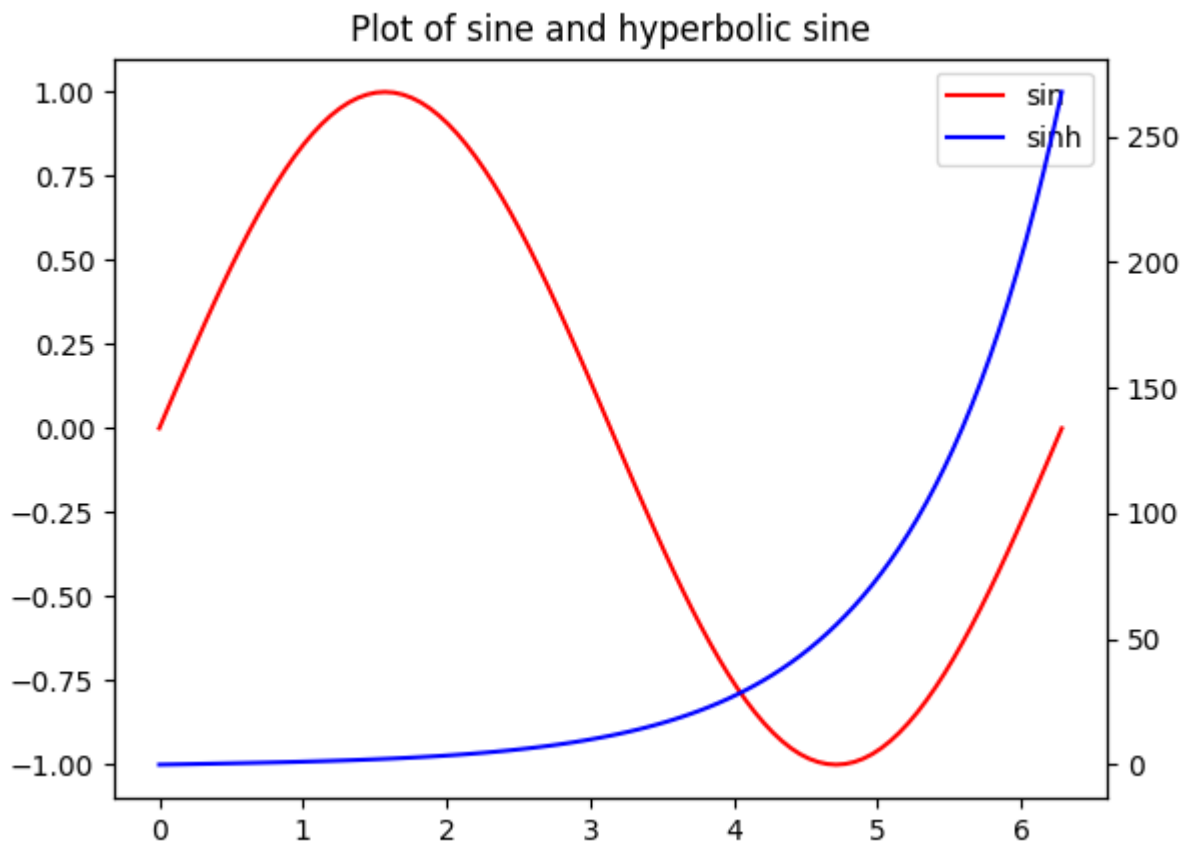
# plot the curves on axes 1, and 2, and get the curve handles
curve1, = ax1.plot(x, y, label="sin", color='r')
curve2, = ax2.plot(x, z, label="sinh", color='b')

# 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()

```

twinyをってのYとなるXをつプロット

ここでは、**twin**メソッドをして、のyをつがxがなるのプロットをしています。また、タイトル、ラベル、グリッド、のり、などのがプロットにされます。

```
# 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.twinx() # 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)

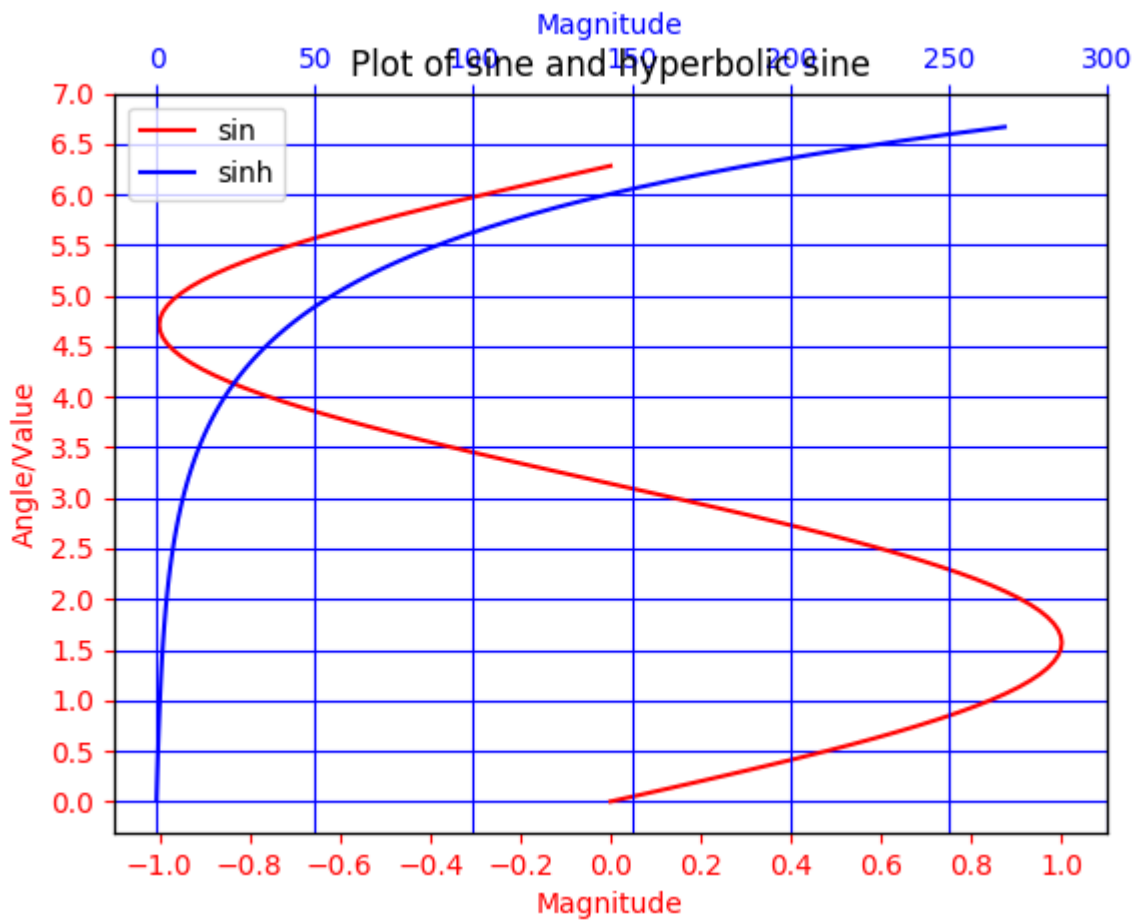
# set y ticks
ax1.set_yticks(ynumbers)
# ax2.set_yticks(ynumbers) # also works

# 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()

```



オンラインでMatplotlibでプロットするをむ <https://riptutorial.com/ja/python/topic/10264/matplotlib>
でプロットする

25: MutableとPythonのおよびHashable

Examples

なものとなもの

Pythonには2のがあります。および。

のオブジェクトはできません。オブジェクトをしようとすると、コピーがされます。

このカテゴリには、、、コンプレックス、、、バイト、タプル、、、およびフーチェンセットがまれます。

このプロパティをするには、`id`みんだものをしてみましょう。これは、パラメータとしてされたオブジェクトののをします。`id`がじ、これはじオブジェクトです。された、これはのオブジェクトです。これはにオブジェクトのメモリアドレスだとわかりますが、それらにはしてください、らはのいからです...

```
>>> a = 1
>>> id(a)
140128142243264
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

さて、1は3ではない...ニュースをる...たぶんそうではない。しかし、よりな、にの、このはしばしばられます。

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
140128123955504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

あある々はそれをすることができます

```
>>> id(stack)
140128123911472
```

いいえ、`stack`でされたをすることはでき`stack`、には、をするしいオブジェクトをしています。なので、いオブジェクトはどこにもきませんので、たちはかれています。のでは、それはもっと

らかだったでしょう

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

この、のをしたいはコピーがです。しかし、それはのタイプのためにとてもらかですか

、がどのようにするかをっていれば、のコードでをいますかそれはですか

```
s = ""
for i in range(1, 1000):
    s += str(i)
    s += ", "
```

ミュータブル

なのオブジェクトはすることができ、そのでされます。のコピーはわれません。

このカテゴリには、リスト、、バイトエリア、セットがまれます。

たちのさな`id`でんでいきましょう。

```
>>> 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
```

すると、*ASCII*データをむバイトをしてポイントをにしますが、バイトはテキストデータをするようにはされていないことをれないでください。

たちはをっていますかたちは`bytearray`をし、それをし、`id`をして、これがされたじオブジェクトであることをすることができます。そのコピーではありません。

もちろん、オブジェクトがにされる、なはよりはるかにいをします。なことに、このプロパティのは、くの、もいときにれられます。

```
>>> c = b
>>> c += b' rocks!'
>>> c
```

```
bytearray(b'StackOverflow rocks!')
```

はい...

```
>>> b
bytearray(b'StackOverflow rocks!')
```

Wait a second ...

```
>>> id(c) == id(b)
True
```

かに。 `c`は`b`コピーではありません。 `c`は`b`です。

ここでは、どのようながによってされているのかをよくしています。このでがうまくいかないのかできますか

```
>>> l1 = [ [] ]*4 # Create a list of 4 lists to contain our results
>>> l1
[[[]], [], [], []]
>>> l1[0].append(23) # Add result 23 to first list
>>> l1
[[23], [23], [23], [23]]
>>> # Oops...
```

としてで

がmutabilityをするがあるなユースケースの1つは、をにすときです。これはにです。これは、スコープにしていなオブジェクトをするをするため、つまりにがあるにをするためです。これは、のをどこでにしなければならないかをするためにもです。

```
>>> 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]
```

ここでは、のパラメータとしての`lin`ローカルでできるとえることです。わりに、`lin`およびじオブジェクト。`a`このオブジェクトはであるため、はインプレースでわれます。つまり、`lin`と`a`でされるオブジェクトがされます。`lin`はにされるはありません。すでにこのオブジェクトへのが`a`ので`a`です。`a`と`b` endはじオブジェクトをします。

これはタプルではじではありません。

```
>>> 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)
```

のに `tin` およびリファレンスじオブジェクト。 `a` しかし、これはのオブジェクトです。はそれをしてしようとするときに、 `tin` で、をえたいオブジェクトをけり、のオブジェクトへのをします。 `a` この、 `tin` すことはであるか、しいオブジェクトがわれます。

```
>>> def yoda(prologue, sentence):
    sentence.reverse()
    prologue += " ".join(sentence)
    return prologue

>>> focused = ["You must", "stay focused"]
>>> saying = "Yoda said: "
>>> yoda_sentence = yoda(saying, focused)
```

`reverse` がインプレースでわれます。

このについてどういすかはありますかリターンはですかコールの、を `saying` ですか `focused` がじパラメータでびされるとどうなりますか

オンラインで [Mutable](https://riptutorial.com/ja/python/topic/9182/mutable) と [Python](https://riptutorial.com/ja/python/topic/9182/mutable) のおよび [Hashable](https://riptutorial.com/ja/python/topic/9182/mutable) をむ

<https://riptutorial.com/ja/python/topic/9182/mutable> と [python](https://riptutorial.com/ja/python/topic/9182/mutable) の-および [hashable](https://riptutorial.com/ja/python/topic/9182/mutable)-

26: os.path

き

このモジュールは、パスにいくつかのなをしています。パスパラメータは、またはバイトとしてすることができます。アプリケーションでは、ファイルをUnicodeとしてすることをおめします。

- `os.path.join`、* `p`
- `os.path.basename`
- `os.path.dirname`
- `os.path.split`
- `os.path.splitext`

Examples

パスをする

2つのパスコンポーネントをにするには、にPythonのosモジュールをインポートし、のようになります。

```
import os
os.path.join('a', 'b', 'c')
```

`os.path`をするは、のプラットフォームにしたりをするため、すべてのオペレーティングシステムでコードのをできることです。

たとえば、Windowsでのこのコマンドのはのようになります。

```
>>> os.path.join('a', 'b', 'c')
'a\b\c'
```

Unix OSの

```
>>> os.path.join('a', 'b', 'c')
'a/b/c'
```

パスからのパス

`os.path.abspath`します。

```
>>> 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'
```

パスコンポーネント

1つのコンポーネントをパスからするには

```
>>> 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' )
```

ディレクトリをする

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

されたパスがする。

されたパスがするかどうかをする

```
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
```

されたパスがディレクトリ、ファイル、シンボリックリンク、マウントポイントなどであるかどうかをしてください。

されたパスがディレクトリかどうかをべる

```
dirname = '/home/john/python'  
os.path.isdir(dirname)
```

されたパスがファイルかどうかをべる

```
filename = dirname + 'main.py'  
os.path.isfile(filename)
```

されたパスがシンボリックリンクかどうかをべる

```
symlink = dirname + 'some_sym_link'
```

```
os.path.islink(symlink)
```

されたパスがマウントポイントかどうかをする

```
mount_path = '/home'  
os.path.ismount(mount_path)
```

オンラインでos.pathをむ <https://riptutorial.com/ja/python/topic/1380/os-path>

27: os モジュール

き

このモジュールは、オペレーティングシステムにするをなでします。

- OS

パラメーター

パラメータ	
パス	ファイルへのパス。パスりは、 <code>os.path.sep</code> によってされます。
モード	のパーミッションを8でえは0700

Examples

ディレクトリをする

```
os.mkdir('newdir')
```

をするがあるは、オプションの`mode`をできます。

```
os.mkdir('newdir', mode=0700)
```

のディレクトリをする

`os.getcwd()`をします。

```
print(os.getcwd())
```

オペレーティングシステムのをする

`os`モジュールは、コードがされているオペレーティングシステムのタイプをするためのインタフェースをします。

```
os.name
```

これは、Python 3でのいずれかをすことができます

- `posix`

- nt
- ce
- java

よりなは `sys.platform` からできます

ディレクトリをする

`path` のディレクトリをし `path` 。

```
os.rmdir(path)
```

ディレクトリをするために `os.remove()` をしないでください。これはファイルであり、ディレクトリでこのをすると `OSError`

シンボリックリンク **POSIX**

によっては、シンボリックリンクのターゲットをするがあります。 `os.readlink` はこれをいます

```
print(os.readlink(path_to_symlink))
```

ファイルのアクセスをする

```
os.chmod(path, mode)
```

`mode` は8でのです。

makedirs - なディレクトリ

ののローカルディレクトリをします。

```
└─ dir1
   └─ subdir1
      └─ subdir2
```

のディレクトリ `dir2` のに `subdir1`、`subdir2` とじサブディレクトリをします。

```
import os

os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

これをすると

```
└─ dir1
   └─ subdir1
      └─ subdir2
└─ dir2
```

```
|— subdir1
|— subdir2
```

dir2は、subdir1ののためになとくにめてされます。

わりに**os.mkdir**をした、dir2はまだしないため、がします。

```
os.mkdir("./dir2/subdir1")
OSError: [Errno 2] No such file or directory: './dir2/subdir1'
```

os.makedirsは、ターゲット・ディレクトリがすでにするはそれをまないのでしよう。した

```
OSError: [Errno 17] File exists: './dir2/subdir1'
```

ただし、をキャッチしてディレクトリがされたことをすることで、これをにできます。

```
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
```

オンラインでosモジュールをむ <https://riptutorial.com/ja/python/topic/4127/osモジュール>

28: Pandas Transform グループのをプリフォームし、をする

Examples

まず、ダミーのデータフレームをします

はnのをすることができ、はmのアイテムをすることができ、アイテムはさらにであるとする

```
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

。

。

さて、々はごとのをえるためにパンダ^{transform}をする

```
# 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
```

グループごとにの

グループごとにサブをす transform を transform

のでは、クライアントごとに1つのがられました。ただし、グループになるをすをすることもできます。

```
# 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
    .groupby(['order_id'])['item']
    # Take the orders dataframe
    # 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

オンラインでPandas Transformグループのをプリフォームし、をするをむ

<https://riptutorial.com/ja/python/topic/10947/pandas-transform-グループのをプリフォームし-をする>

29: Pandasをしたデータファイルの、サブセットおよび

き

このセクションでは、パンダをしたデータファイルのみみ、サブ、およびきしのなコードをします。

Examples

Pandasをしてデータファイルをインポート、サブセット、およびきむためのコード

```
# Print the working directory
import os
print os.getcwd()
# C:\Python27\Scripts

# Set the working directory
os.chdir('C:/Users/generall/Documents/simple Python files')
print os.getcwd()
# C:\Users\generall\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)

```

オンラインでPandasをしたデータファイルの、サブセットおよびをむ

<https://riptutorial.com/ja/python/topic/8854/pandasをしたデータファイルの-サブセットおよび>

30: pipPyPIパッケージマネージャ

き

pipはPython Package Indexでもくわれているパッケージマネージャです。Pythonのバージョンではデフォルトでインストールされます。

- pip <command> [options] <command>はのいずれかです。
 - インストール
 - パッケージをインストールする
 - アンインストール
 - パッケージのアンインストール
 - フリーズする
 - インストールされたパッケージをでする
 - リスト
 - インストールされているパッケージの
 - ショー
 - インストールされたパッケージにするをする
 - サーチ
 - パッケージのPyPIをする
 - ホイール
 - あなたのからホイールをする
 - ジップ
 - 々のパッケージを
 - する
 - 々のパッケージをする
 - バンドル
 - pybundlesをする
 - けて
 - コマンドのヘルプをする

によっては、pipはネイティブコードのコンパイルをします。Linuxでは、PythonはシステムでなCコンパイラをにします。WindowsでなVisual Studio / Visual C ++のバージョンについては、のをしてくださいしいバージョンはしません。

Pythonバージョン	Visual Studioバージョン	Visual C ++バージョン
2.6 - 3.2	Visual Studio 2008	Visual C ++ 9.0
3.3 - 3.4	Visual Studio 2010	Visual C ++ 10.0
3.5	Visual Studio 2015	Visual C ++ 14.0

Examples

パッケージをインストールする

`SomePackage`というパッケージのバージョンをインストールするには

```
$ pip install SomePackage
```

のバージョンのパッケージをインストールするには

```
$ pip install SomePackage==1.0.4
```

パッケージにインストールするバージョンをするには

```
$ pip install SomePackage>=1.0.4
```

コマンドがLinux / Unixでパーミッションエラーをした、コマンドに`sudo`をします

ファイルからインストールする

```
$ pip install -r requirements.txt
```

ファイルのは、インストールするものをし、`pip`インストールのとに、ファイルののはのとおりです。 [ファイルの](#)。

パッケージをインストールした、`freeze`コマンドをしてチェックすることができます

```
$ pip freeze
```

パッケージのアンインストール

パッケージをアンインストールするには

```
$ pip uninstall SomePackage
```

``pip``をってインストールされたすべてのパッケージをするには

インストールされたパッケージをするには

```
$ pip list
# example output
```

```
docutils (0.9.1)
Jinja2 (2.6)
Pygments (1.5)
Sphinx (1.1.2)
```

いパッケージをし、なバージョンをするには

```
$ pip list --outdated
# example output
docutils (Current: 0.9.1 Latest: 0.10)
Sphinx (Current: 1.1.2 Latest: 1.1.3)
```

パッケージのアップグレード

ランニング

```
$ pip install --upgrade SomePackage
```

パッケージ`SomePackage`とそのすべてのをアップグレードします。また、`pip`はアップグレードのにいバージョンのパッケージをにします。

`pip`をアップグレードするには、

```
$ pip install --upgrade pip
```

Unixまたは

```
$ python -m pip install --upgrade pip
```

Windowsマシン。

Linuxでいパッケージをすべてする

`pip`は、ユーザーがにすべてのいパッケージをできるようにするフラグがまれていません。しかし、これは、Linuxでコマンドをにすることでできます。

```
pip list --outdated --local | grep -v '^-e' | cut -d = -f 1 | xargs -n1 pip install -U
```

このコマンドは、ローカルの`virtualenv`のすべてのパッケージをりし、いものかどうかをチェックします。そのリストから、パッケージをし、`pip install -U`コマンドにパイプします。このプロセスのに、すべてのローカルパッケージをするがあります。

Windowsでいパッケージをすべてする

`pip`は、ユーザーがにすべてのいパッケージをできるようにするフラグがまれていません。ただし、これは、Windowsでコマンドをにすることでできます。

```
for /F "delims= " %i in ('pip list --outdated --local') do pip install -U %i
```

このコマンドは、ローカルのvirtualenvのすべてのパッケージをりし、いものかどうかをチェックします。そのリストから、パッケージをし、`pip install -U`コマンドにパイプします。このプロセスのに、すべてのローカルパッケージをするがあります。

システムのすべてのパッケージの**requirements.txt**ファイルをする

pipは**freeze**オプションをすることによって**requirements.txt**ファイルをするのにちます。

```
pip freeze > requirements.txt
```

これにより、システムにインストールされているすべてのパッケージとそのバージョンのリストが、のフォルダの**requirements.txt**というのファイルにされます。

の**virtualenv**でのみパッケージの**requirements.txt**ファイルをする

pipは**freeze**オプションをすることによって**requirements.txt**ファイルをするのにちます。

```
pip freeze --local > requirements.txt
```

--localパラメータは、**virtualenv**にローカルにインストールされているパッケージとバージョンのリストのみをしします。グローバルパッケージはリストされません。

pipでの**Python**バージョンをする

Python 3とPython 2のがインストールされている、pipでするPythonのバージョンをできます。これは、パッケージがPython 2または3のみをサポートしている、またはでテストしたいにです。

Python 2のパッケージをインストールするは、のいずれかをします。

```
pip install [package]
```

または

```
pip2 install [package]
```

Python 3のパッケージをインストールするは、のようにします。

```
pip3 install [package]
```

のPythonインストールにパッケージのインストールをびすには、のようにします。

```
\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
```

OS-X / Linux / Unixプラットフォームでは、PythonのシステムバージョンシステムをにするアップグレードとPythonのユーザーバージョンとのにすることがです。あなたは、あなたがアップグレードしようとしているものにして、これらのコマンドのに `sudo` をつけてパスワードをするがあります。

Windowsの、いくつかのPythonインストール、にのパッケージのであるインストールは、モードでしているコマンドウィンドウからアップグレードするのあるシステムディレクトリにインストールされるがあります。 `python -c"import sys;print(sys.path);"`などのコマンドをして、どのPythonインストールをアップグレードしようとしているかをするには `python -c"import sys;print(sys.path);"` または `py -3.5 -c"import sys;print(sys.path);"` `pip --version` しようとしているパイプをすることもできます

Windowsの、Python 2とPython 3のがインストールされていて、パスにPython 3が3.4よりきいは、おそらくシステムパスに `python` ランチャー `py` があるでしょう。に、のようなトリックをうことができます

```
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
```

のバージョンのPythonをしているは、pythonのバージョンとするパッケージのをできるように、`python virtualenv` または `venv virtualenv` についておみになることをくおめします。

としてピップにまだっていないパッケージをインストールする

くのなPythonパッケージは、まだPython Package Indexではとしてできませんが、まだうまくインストールされています。しかし、Windowsのいくつかのパッケージでは、ろしい `vcvarsall.bat` がつかりませんでした。

は、インストールしようとしているパッケージにCまたはC++のがまれていて、pythonパッケージのインデックス、`pypi`、およびビルドになツールチェーンをたないウィンドウのプリビルドホイールとしてできないということですそのような。

もなえは、[Christoph Gohlke](#)のれたサイトにアクセスし、なライブラリのなバージョンをつけることです。パッケージのなことで `-cp NN` にをける-あなたは `-win32`-めるがあり、64ビットのpythonをしている、それは `-win_amd64` をまなければならいとしてもをWin64のの32ビットのpythonをしている、すなわち、パイソンのバージョンとするがあります-Pythonのバージョンがするがあります。つまり、Python 34の、ファイルには `-cp 34`-などがまれていなければなりません。これはにpipがpypiサイトでうです。

あるいは、あなたがっているPythonのバージョンのなWindowsキット、インターフェイスをビルドしようとしているパッケージのライブラリ、おそらくPythonのバージョンのためのPythonヘッダなどをするがあります。

Python 2.7はVisual Studio 2008をし、Python 3.3および3.4はVisual Studio 2010をし、Python 3.5+はVisual Studio 2015をします。

- MicrosoftのWebサイトから「[Python 2.7のVisual C ++コンパイラパッケージ](#)」をインストールするか、
- マイクロソフトのWebサイトから「[Windows 7および.NET Framework 4Windows SDK](#)」v7.1をインストールするか、
- [Visual Studio 2015 Community Edition](#) またはそのバージョンがリリースされたときにインストールするをインストールして、**CC ++**サポートをインストールするオプションをデフォルトではなくするようにします。ダウンロードとインストールに **8**かかることがありますこれらのオプションがあるのでされていることをしてください。

に、ヘッダファイルをつけ、なパッケージがリンクしているライブラリのリビジョンにわせてなにダウンロードするがあります。

に、pipにビルドをさせることもできます。もちろん、パッケージにまだしていないがあるは、それらのヘッダファイルもつけるがあります。

のピピやクリストフのサイトでも、あなたがしているパッケージののバージョンは、なPythonか、あなたのプラットフォームとPythonバージョンににされていて、それらをしているがありますあなたのパッケージがになるまで、つけました。に、バージョンのPythonをしているは、パッケージがしをえて、のパッケージがになプロジェクトのは、しいPythonをしなければならないかもしれません。また、パッケージのソースサイトをチェックして、めビルドされている、またはなPythonとしてなフォークされたバージョンがあるかどうかをし、なをするパッケージをすこともできます。にされている [は](#)、6でされていない [PIL](#)のわりにドロップされ、Python 3ではできません。

その、このをえているには、パッケージのバグトラッカーにアクセスして、パッケージメンテナがののためにピピのをすることをにするチケットをするか、プラットフォームとPythonのみわせ、これがわれれば、パッケージのメンテナは、々がしているのあるみわせをしていることにかないことがあります。

プレリリースのインストールにする

Pipは[Semantic Versioning](#)のルールにい、デフォルトではリリースのパッケージよりもリリースされたパッケージをします。したがって、のパッケージがv0.98としてリリースされていて、リリースv1.0-rc1ある、`pip install`デフォルトのはv0.98をインストールすることv0.98 - リリースをインストールするは、にでテストするには、`--pip install --pre package-name`または`--pip install --pre --upgrade package-name`をしてにすることができます。くの、プレリリースまたはリリースは、すべてのプラットフォームとバージョンのみわせにしてホイールがされていないがあるため、のにするがさらにくなります。

バージョンのインストールにする

また、pipを使ってgithubやのからパッケージのをインストールすることもできます。そのようなコードはであるため、ホイールをみむことはほとんどありませんので、なパッケージにはビルドツールがです。いつでもすることができます。そのため、ユーザーはにそのようなパッケージをインストールすることをくします。

そのようなインストールには3つのオプションがあります。

1. スナップショットをダウンロードすると、ほとんどのオンラインバージョンシステムでは、コードのスナップショットをダウンロードすることができます。これはでダウンロードし、`pip install path / to / downloaded / file`で `pip install` ことができます。ほとんどのではpipがキャッシュなどへのアンパックをすることにしてください。
2. あなたがダウンロードしてあなたのために `pip install` インストールさせるようにしましょう
`pip install URL / of package / repository` - これには `--trusted-host`、`--client-cert` および `または --proxy` フラグをするがありますにではくします。えは

```
> 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

```

URLの`git+`にしてください。

3. してリポジトリをクローン`git`、`mercurial`またはのなツール、もしくはDVCSツールを、そして`pip install` クローン/レポ/に/パスを -こののプロセスがどの`requires.txt`ファイルとビルドとセットアップのをし、 ですることができますあなたのクローンされたリポジトリへのディレクトリと`pip install -r requires.txt`し、その`python setup.py install`とじをるために。このアプローチのきなは、のクローンはスナップショットのダウンロードよりもがにかかるかもしれませんが、`git git pull origin master`はのものにでき、のバージョンにエラーがあるは`pip uninstall package-name`に`git checkout` コマンドをして、リポジトリのをのバージョンにしています。

オンラインで`pipPyPI`パッケージマネージャをむ <https://riptutorial.com/ja/python/topic/1781/pip-pypi>パッケージマネージャ

31: PostgreSQL

Examples

PostgreSQLはにされたオープンソースデータベースです。 `psycopg2`モジュールをして、データベースにしてクエリをすることができます。

pipをったインストール

```
pip install psycopg2
```

な

のようにされたデータベース`my_database`テーブル`my_table`があるとします。

id	ファーストネーム	
1	ジョン	Doe

`psycopg2`モジュールをして、のでデータベースでクエリをすることができます。

```
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')]
```

オンラインでPostgreSQLをむ <https://riptutorial.com/ja/python/topic/3374/postgresql>

32: py.test

Examples

py.testの

py.testは、Pythonでなくいくつかのサードパーティのテストライブラリの1つです。それはpipをしてインストールすることができます

```
pip install pytest
```

テストするコード

projectroot/module/code.pyをテストしているとします。

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

テストコード

projectroot/tests/test_code.pyテストファイルをしします。このファイルは、テストファイルとしてされるには、test_まるかあります。

```
# projectroot/tests/test_code.py
from module import code

def test_add():
    assert code.add(1, 2) == 3
```

テストの

projectroot 々はに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
=====
```

したテスト

したテストは、がうまくいかなかったかにするなをします

```
# projectroot/tests/test_code.py
from module import code

def test_add__failing():
    assert code.add(10, 11) == 33
```

```
$ 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
=====
```

テストへの

よりなテストでは、テストするコードをするに、がなことがあります。これをテストでうことはですが、なテストがすぎるため、セットアップがどこでし、テストがまるかをることがしくなります。また、さまざまなテストでしたセットアップコードをすることもできます。

たちのコードファイル

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
        self.bar = 2
```

たちのテストファイル

```
# 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
```

これらはになですが、`Stuff`オブジェクトにもっとくのがなはいにくくなります。たちのテストケースのにはしたコードがあることがわかりますので、まずそれをのにリファクタリングしましょう。

```
# 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
```

これはくえますが、たちはまだ`my_stuff = get_prepped_stuff()`びしでテストを`my_stuff = get_prepped_stuff()`しています。

のpy.testの

フィクスチャは、テストセットアップのはるかにでなバージョンです。たちがここでしているにくのことをうことができますが、に1ステップずつめていきます。

まず、`get_prepped_stuff`とばれるに`prepped_stuff`。フィクスチャがでテストでどのようにされるのかで、ではなくでフィクスチャのをけるがあります。 `@pytest.fixture`は、こののをではなくフィクスチャーとしてするがあることをします。

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

ここで、フィクスチャをするようにテストをするがあります。これは、フィクスチャとにするパラメータをにすることによってわれま。 `py.test`がされると、テストをするにフィクスチャをし、フィクスチャのりをそのパラメータをしてテストにします。フィクスチャはをすはなく、リソースをびす、ファイルシステムにアレンジする、データベースにをれる、セットアップになテストをうなどののをうことができます。

```
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
```

これでなぜにをけたのかかります。 `my_stuff = prepped_stuff`はかなりにたないので、わりに `prepped_stuff` しましょう。

```
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
```

はをしていますフィクスチャのをすることでテスト・モジュールごと、またはテスト・ファンクションごとに1だけする、のフィクスチャをするフィクスチャをし、フィクスチャをパラメータすることでそのフィクスチャをするテストは、フィクスチャにえられたパラメータにして1、されます、それらをびすモジュールからをみるフィクスチャ...のように、フィクスチャはのセットアップよりもくのパワーとをえています。

テストのクリーンアップ。

コードがきくなり、Stuffオブジェクトがなクリーンアップをとしているとしましょう。

```
# 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
```

すべてのテストのにクリーンアップをびすためのコードをすることができますが、フィクスチャはこれをうためのよりいをしします。フィクスチャにをしてファイナライザとしてすると、フィクスチャをしたテストがしたにファイナライザのコードがびされます。フィクスチャのスコープがのモジュールやセッションのようなよりきい、スコープのすべてのテストがしたでファイナライザがされるため、モジュールのがした、またはテストセッションの。

```
@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
```

でファイナライザをすることは、になフィクスチャをしているには、でかりにくいことがあります。わりに、 まりフィクスチャをして、がめるフローでじことをうことができます。ののいは、わりにすることである`return`々がする`yield`のセットアップがし、コントロールがにすべてのクリーンアップコードをし、テストにくべきので`yield`。また、それを`yield_fixture`として`yield_fixture`し、 `yield_fixture`それをどのようにうかをとっているようにします。

```
@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()
```

そしてそれは、テスト・フィクスチャーのです。

については、 [のpy.test fixtureのドキュメント](#)と[のyield fixtureのドキュメント](#)をしてください。

オンラインでpy.testをむ <https://riptutorial.com/ja/python/topic/7054/py-test>

33: Py2Neoを使ったNeo4jとCypher

Examples

インポートと

```
from py2neo import authenticate, Graph, Node, Relationship
authenticate("localhost:7474", "neo4j", "<pass>")
graph = Graph()
```

Neo4jデータベースがlocalhost7474にし、なをとっていることをするがあります。

graphオブジェクトはあなたのPythonコードのりのneo4jインスタンスへのインタフェースです。むしろこれをグローバルにすることにします。クラスの__init__メソッドでそれをするがあります。

Neo4j グラフへのノードの

```
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()
    [...]
```

グラフにノードをするのはにです。グラフgraph.merge_oneは、するアイテムをぐのです。スクリプトを2すると、2にタイトルがされ、じのしいノードはされません

timestampはでなくてはならず、ではありません。なぜなら、neo4jはにデータをとっていないからです。これは、を'05 -06-1989'としてするとべえのがします

article.push()は、にをneo4jにコミットするびしです。このステップをれないでください。

Neo4j グラフへのの

```
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`は、をけるためにです。しかしそれはかなりなです。リレーションシップは、なケースであるものにです。

クエリ1 ニュースタイトルのオートコンプリート

```
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
```

これは、`text` でまるプロパティ `name` をつすべてのノードをするサンプルのサイファークエリ `text`。

クエリ2 のにニュースをする

```
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
```

このクエリをすると、によって `(l)` されているすべてのニュース `(n)` をつけることができます。

Cypher クエリサンプル

のとともにのにするをえます

```
MATCH (n)-[]->(l)
where l.name='Donald Trump'
RETURN n.date,count(*) order by n.date
```

なくとも5つのトータルリレーションシップノードをつトランプとじニュースにされているの/をします。

```
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
```

オンラインでPy2NeoをつたNeo4jとCypherをむ

<https://riptutorial.com/ja/python/topic/5841/py2neoをつたneo4jとcypher>

34: pyautoguiモジュール

き

pyautoguiは、マウスとキーボードをするためのモジュールです。このモジュールは、に、マウスのクリックやキーボードのプレスをするために使われます。マウスの、の0,0はからまります。あなたがコントロールできないは、マウスカーソルをすばやくにすると、マウスとキーボードのがPythonからけり、あなたにします。

Examples

マウス

これらは、マウスをするためのなマウスのです。

```
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.
```

キーボード

これらは、キーをすことをするなキーボードのです。

```
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.
```

スクリーンショットと

これらのをすると、スクリーンショットをったり、のとをさせることができます。

```
.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.
```

オンラインでpyautoguiモジュールをむ <https://riptutorial.com/ja/python/topic/9432/pyautogui> モジュール

35: PyInstaller - Pythonコードの

- `pyinstaller [options]スクリプト[script ...] | specfile`

PyInstallerは、すべてのとともにPythonアプリケーションを1つのパッケージにバンドルするためにされるモジュールです。ユーザーは、Pythonインタプリタやモジュールなしでパッケージアプリをできます。numpy、Django、OpenCvなどのくのパッケージをしバンドルしています。

えておくべきなポイント

- PyinstallerはPython 2.7およびPython 3.3+をサポートしています
- PyinstallerはWindows、Linux、Mac OS Xでテストみです。
- クロスコンパイラではありません。WindowsアプリケーションはLinuxにパッケージすることはできません。PyInstallerをWindowsでしてWindowsのアプリケーションをバンドルするがあります

[ホームページ](#)

Examples

インストールとセットアップ

PyinstallerはのPythonパッケージです。それはピップをしてインストールすることができます

```
pip install pyinstaller
```

Windowsでのインストール

Windowsの、[pywin32](#)または[pypiwin32](#)はです。はpyinstallerをpipでインストールするとにインストールされます。

Mac OS Xでのインストール

PyInstallerは、のMac OS XでされているデフォルトのPython 2.7でします。Pythonのそののバージョンをするや、PyQT、Numpy、Matplotlibなどのなパッケージをするは、[MacPorts](#)または[Homebrew](#)のいずれかです。

アーカイブからのインストール

pipができないは、[PyPI](#)からされたアーカイブをダウンロードしてください。

バージョンをテストするには、[PyInstaller Downloads](#)ページのブランチからアーカイブをダウンロードしてください。

アーカイブをし、`setup.py`スクリプトをします。PyInstallerをインストールまたはアップグレード
`python setup.py install`、`python setup.py install with administrator`をします。

インストールの

`pyinstaller` コマンドは、インストールが完了した、すべてのプラットフォームのシステムパスにするがあります。

コマンドラインで `pyinstaller --version` としてします。これにより、のバージョンの `pyinstaller` がされます。

Pyinstaller の

もなユースケースでは、ファイルがあるディレクトリにし、のようにします。

```
pyinstaller myfile.py
```

Pyinstaller はファイルをし、をします。

- じディレクトリにあるファイル **myfile.spec** `myfile.py`
- `myfile.py` とじディレクトリにあるビルドフォルダ
- `myfile.py` とじディレクトリにある **dist** フォルダ
- ビルドフォルダのログファイル

バンドルされたアプリは **dist** フォルダにあります

オプション

`pyinstaller` にはいくつかのオプションがあります。オプションのなリストは [ここ](#) でつけることができます。

バンドルされると、あなたのアプリは `'dist \ myfile \ myfile.exe'` をいてできます。

1つのフォルダにバンドルする

`pyInstaller` のをバンドルするオプションをせずにした `myscript.py`、デフォルトのは、のフォルダはある `myscript` のむ `myscript myscript.exe` なすべてののにウィンドウに。アプリは、フォルダを zip ファイルにしてすることができます。

1つのフォルダモードは、オプション `-D` または `--onedir` をしてにできます

```
pyinstaller myscript.py -D
```

1つのフォルダにバンドルすることのきなのは、をデバッグするがだということです。いずれかのモジュールがインポートにしたは、フォルダをべてできます。にのがじられます。コードにいくつかのがあっても、するがまったくじであれば、はファイルはフォルダよりもさいをすることができます。

こののは、ユーザーがのファイルのからファイルをするがあることです。また、ユーザーは、アプリケーションがにしなくなるのあるのファイルを/することができます。

ファイルへのバンドル

```
pyinstaller myscript.py -F
```

のファイルをするオプションは、`-F`または`--onefile`です。これにより、プログラムが1つの `myscript.exe` ファイルにバンドルされます。

ファイルファイルは、フォルダバンドルよりもです。らはまた、デバッグするのがしいです。

オンラインでPyInstaller - Pythonコードのをむ

<https://riptutorial.com/ja/python/topic/2289/pyinstaller-----pythonコードの>

36: Python 2からPython 3への

き

ほとんどのとはなり、Pythonは2つのメジャーバージョンをサポートしています。Python 3がリリースされた2008、くのがをいしましたが、くはそうしていませんでした。をするために、このセクションではPython 2とPython 3のなについてします。

サポートされているPythonのバージョンは2.7Python 2と3.6Python 3です。さらに、バージョン3.3および3.4は、ソースでセキュリティプログラムをします。

Python 2.7はほとんどのバージョンのPythonとがあり、Pythonのほとんどの1.xおよび2.xバージョンのPythonコードをせずにできます。これはにであり、なパッケージをめています。また、CPythonはとなされ、セキュリティとバグののみをけります。CPythonのは、このバージョンのを2020にしようとしています。

[Python Enhancement Proposal 373](#)によれば、2016625、Python 2ののリリースはされていませんが、2020まではバグやセキュリティアップデートがサポートされます。2020のなはPythonの2

Python 3は、がのにあったにするためににをつた。Python 3はしいとをけります。これは、がするののバージョンです。

Python 3.0のリリースからのバージョンまでのに、Python 3のいくつかのはPython 2.6にバックポートされ、Python 3ののはPython 2とのがされました。したがって、Python 2とPython 3ので、のインポートとなモジュール **6**のようなをとてするPython。

のインポートは、モジュールのめにあるがあります。

```
from __future__ import print_function
# other imports and instructions go after __future__
print('Hello world')
```

`__future__` モジュールのについては、[Pythonドキュメントのページ](#)をしてください。

[2to3ツール](#)は、Python 2.xコードをPython 3.xコードにするPythonプログラムです [Pythonのドキュメント](#)もしてください。

パッケージ**6**はPython 2/3とののためのユーティリティをします

- をしたライブラリへのされたアクセス
- /ユニコードタイプの
- されたメソッドまたはがされたメソッドの

Python 2とPython 3のについては、[ここででき](#)ます。

Examples

ステートメントと

Python 2では、`print`はステートメントです

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)"
```

Python 3では、`print()`はのキーワードをつです

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)"
```

には、のパラメータがあります。

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep`は、すオブジェクトをとってするものです。えば

```
print('foo', 'bar', sep='~') # out: foo~bar
print('foo', 'bar', sep='.') # out: foo.bar
```

`end`は`print`ステートメントのわりのにくものです。えば

```
print('foo', 'bar', end='!') # out: foo bar!
```

じにされます `print`をする、の

```
print('foo', end='~')
print('bar')
# out: foo~bar
```

このために、`print` はPython 2.6でもできます。ただし、`print` ステートメントの `print` がにされていないりすることはできません

```
from __future__ import print_function
```

これは、`flush` パラメータがないをいて、Python 3とまったくじです。

については、PEP [3105](#)をしてください。

バイトとUnicode

Python 2.x 2.7

Python 2には、`str` のバイトと`str`のテキスト `unicode` からなる2のがあります。

Python 2では、`str`のオブジェクトはにバイトシーケンスですが、テキストデータとバイナリデータのによくされます。

リテラルはバイトとしてされます。

```
s = 'Cafe'    # type(s) == str
```

2つのがあります。Unicodeテキストリテラルをにするには、リテラルに`u`

```
s = u'Café'    # type(s) == unicode
b = 'Lorem ipsum' # type(b) == str
```

あるいは、モジュールのリテラルがUnicodeテキストリテラルをするようにすることもできます。

```
from __future__ import unicode_literals

s = 'Café'    # type(s) == unicode
b = 'Lorem ipsum' # type(b) == unicode
```

がUnicodeかバイトかどうかをするには、のようになります

```
isinstance(s, basestring)
```

Python 3.x 3.0

Python 3では、`str`はUnicodeテキストです。

```
s = 'Cafe'          # type(s) == str
s = 'Café'          # type(s) == str (note the accented trailing e)
```

さらに、Python 3では、バイナリの「プロブ」やエンコードにしないファイルへのきみにした `bytes` オブジェクトがされました。バイトオブジェクトをするには、`b` をリテラルにするか、の

`encode` メソッドをびします。

```
# Or, if you really need a byte string:
s = b'Cafe'          # type(s) == bytes
s = 'Café'.encode() # type(s) == bytes
```

がかどうかをテストするには、のようになります。

```
isinstance(s, str)
```

Python 3.x 3.3

Python 2とPython 3のコードベースのをにするために、リテラルに`u`けることもできます。
Python 3では、すべてのがデフォルトでUnicodeなので、リテラルのに`u`てもありません。

```
u'Cafe' == 'Cafe'
```

Python 2ののUnicodeの`ur`はサポートされていません。

```
>>> ur'Café'
File "<stdin>", line 1
    ur'Café'
      ^
SyntaxError: invalid syntax
```

Python 3 `text str` オブジェクトを`encode`、そのテキストの`bytes`にするがあることにしてください。
このメソッドのデフォルトのエンコーディングはUTF-8です。

`decode` をって `bytes` オブジェクトに、それがすUnicodeテキストをねることができます

```
>>> b.decode()
'Café'
```

Python 2.x 2.6

`bytes` タイプはPython 2と3のにし`bytes`が、`unicode`はPython 2にのみします。Python 2でPython 3ののUnicodeをするには、コードファイルのにのコードをします。

```
from __future__ import unicode_literals
print(repr("hi"))
# u'hi'
```

Python 3.x 3.0

もう1つのないは、Python 3のインデックスバイトが`int`になることです。

```
b"abc"[0] == 97
```

1のサイズでスライスすると、さが1バイトのオブジェクトになります。

```
b"abc"[0:1] == b"a"
```

さらに、Python 3はユニコードでいくつかのなをしします。つまり、Python 2でバイトをしします。たとえば、[の](#)がされました。

```
# -*- 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.
# .agnaŁ zsakuŁ si eman ym ,iH
# .agnaŁ zsaku♦♦ si eman ym ,iH

# Output in Python 3
# Hi, my name is Łukasz Langa.
# .agnaŁ zsakuŁ si eman ym ,iH
# .agnaŁ zsakuŁ si eman ym ,iH
```

/ は、にすると、Python 3とPython 2ではがなります。

Python 3でをのけると、 x / y はのをし `__truediv__` メソッドを、のをしします。、パイソン2におけるのをすなまたをるとしてもらわれるのにかってダウンをめます。

えば

コード	Python 2の	Python 3の
3 / 2	1	1.5
2 / 3	0	0.6666666666666666
-3 / 2	-2	-1.5

ゼロへのめのるいは、[Python 2.2](#)ではされましたが、のためにPython 2.7にっており、Python 3ではされています。

Python 2でのをるにはフロアのめなし、オペランドの1つをでしします。のPython 2で0をえる $2/3$ は、 $2 / 3.0$ または $2.0 / 3$ または $2.0/3.0$ としてされ、`0.6666666666666666`

コード	Python 2の	Python 3の
3.0 / 2.0	1.5	1.5
2 / 3.0	0.6666666666666666	0.6666666666666666
-3.0 / 2	-1.5	-1.5

フロア `//` もあり、のバージョンでじようにします。もいにりげられます。ですとがされますがどちらのバージョンでも、`//`は`__floordiv__`マップされ`__floordiv__`。

コード	Python 2の	Python 3の
<code>3 // 2</code>	1	1
<code>2 // 3</code>	0	0
<code>-3 // 2</code>	-2	-2
<code>3.0 // 2.0</code>	1.0	1.0
<code>2.0 // 3</code>	0.0	0.0
<code>-3 // 2.0</code>	-2.0	-2.0

`operator`モジュールのネイティブをして、のまたはフロアをにすることができます。

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25          # equivalent to `/` in Python 3
assert floordiv(10, 8) == 1            # equivalent to `//`
```

ですが、すべてののをするのはです。`/`のをするがましいことがい。なは`from __future__ import division`をモジュールののステートメントとしてすることによって、なのるいをすることです。

```
# needs to be the first statement in a module
from __future__ import division
```

コード	Python 2の	Python 3の
<code>3 / 2</code>	1.5	1.5
<code>2 / 3</code>	0.6666666666666666	0.6666666666666666
<code>-3 / 2</code>	-1.5	-1.5

`from __future__ import division`は、`/`が`__future__ import`をむモジュールでのみのをしていることをするので、すべてのしいモジュールでそれをにしないなはありません。

のいくつかのプログラミングは、Pythonがののにめるのではなく、ゼロにりてるtruncationようにしますつまり、それらのでは`-3 / 2 == -1`。このは、コードのまたはにをくがあります。

のオペランドにする `from __future__ import division`わりに、のシンボル`/`をし、オペランドのなくとも1つがfloat `3 / 2.0 == 1.5`ことをすることができます。しかし、これはいとみなすことがで

きます。 `average = sum(items) / len(items)` をいて `float` にの1つをキャストするのをやるのはです。あなたがたでテストするはさらに、このようなケースはしばしば、ええ、テストにやる `float` が、ける `int` で。さらに、じコードがPython 3でされている、 `3 / 2 == 1` がTrueであるとされるプログラムはしくしません。

がPython 3でされたといスタイルのをやるについては、 [PEP 238](#) をしてください。

のについては、「 [な](#) 」のトピックをしてください。

Reduceはもはやビルトインではありません

Python 2では、 `reduce` はみみまたは `functools` パッケージバージョン2.6からできますが、Python 3では `reduce` は `functools` からのみできます。ただしの `reduce` Python2とのpython3のではじであり、である `reduce(function_to_reduce, list_to_reduce)` 。

として、やるのそれぞれをやることによってリストのをのにらすことをえてみましょう。ここでは、 `operator` ライブラリの `truediv` をします。

Python 2.xではのようにです

Python 2.x 2.3

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

Python 3.xでは、このはしになりました。

Python 3.x 3.0

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

また `from functools import reduce` を `from functools import reduce` て、で `reduce` をびさないように `reduce` こともできます。

と `xrange` のい

Python 2では、 `range` はリストをしますが、 `xrange` はな `xrange` オブジェクトをします。これはのみみシーケンスとなり、スライシングをサポートせず、 `index` メソッドも `count` メソッドもありません。

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
```

Python 3では、`xrange`が`range`シーケンスにされ、`range`オブジェクトがされました。`xrange`タイプはありません

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
```

さらに、Python 3.2、`range`はスライシング、`index`、および`count`もサポートしてい

```
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
```

リストのわりになシーケンスをするは、インタプリタがリストのメモリをりてて、それをするがないことです。

Python 2.x 2.3

```
# range(1000000000000000000)
# The output would be:
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# MemoryError

print(xrange(1000000000000000000))
# Out: xrange(1000000000000000000)
```

のはにはましいので、Python 3ではがされました。Python 3でリストをしたいは、`range`オブジェクトで`list()`コンストラクタをうだけです

Python 3.x 3.0


```
print(list(range(1, 10)))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python 2.xとPython 3.xのバージョンのをするために、`future`のパッケージから`builtins`モジュールをして、と のをすることができます。

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
```

`future`ライブラリの`range`は、Python 3.2のみメソッドとに、すべてのPythonバージョンでスライス、`index`、および`count`サポートしています。

イテラブルの

Python 3.x 3.0

Python 3では、なのなをらずになものをアンパックすることができ、なのわりをするをつことさえます。そのために、のリストをするをします。これは、のにアスタリスクをくことによってわれます。たとえば、`list`アンパック

```
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
```

*`variable`をする、のタイプがリストでなくても、`variable`はにリストになります。のリストののにじて、0のをむことがあります。

```
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
```

に、 `str`

```
begin, *tail = "Hello"
print(begin)
# Out: 'H'
print(tail)
# Out: ['e', 'l', 'l', 'o']
```

`date`のアンパックの。 `_`は、このではいてとしてされていますたちは`year`だけにがあります

```
person = ('John', 'Doe', (10, 16, 2016))
*_, (*_, year_of_birth) = person
print(year_of_birth)
# Out: 2016
```

`*`はのアイテムをべるので、りてでじ`iterable`にして2つの`*`つことはできません - のアンパックに
るのと2のアンパックのはかりません

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
```

Python 3.x 3.5

これまでは、りてのについてしました。 `*`と`**`は[Python 3.5でされました](#)。1つのでのアンパック
をうことがになりました。

```
{*range(4), 4, *(5, 6, 7)}
# Out: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x 2.0

`iterable`をのにすることもです

```
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

をすると、する2つの`**`されます [PEP 448](#)。

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
# Out: {'x': 1, 'y': 2, 'z': 3}
```

これにより、いをきしてをマージすることができます。

```
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は、のタプルのアンパックをしました。したがって、はPython 3ではしません

```
# 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
```

なについては、PEP [3113](#)をしてください。

のと

これは、Pythonの2で、カンマにし、のraise^びexcept ラインを

Python 2.x 2.3

```
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

Python 3では、がされ、かつことas キーワードでさえられます。

```
try:
    raise IOError("input/output error")
except IOError as exc:
    print(exc)
```

をつために、Python 3のでもPython 3のをできるため、のバージョンとがあるのないすべてのしいコードにするがあります。

Python 3.x 3.0

Python 3では、[の](#)もされています。そこで、こののがいくつかであることをえることができます。えば

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from e
```

`except` であるは `DatabaseError` ですが、のはそのの `__cause__` としてマークされています。トレースバックがされると、のトレースバックにされます。

```
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')
```

なをせずに `except` ブロックをげた

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}')
```

トレースバックは

```
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

どちらもPython 2.xではサポートされていません。 `except` ブロックでのがした、のとそのトレースバックはわれます。のためにのコードをできます。

```
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

にスローされたを "れる"には、 `raise from None` します。

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from None
```

トレースバックはに

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
DatabaseError('Cannot open database.db')
```

または、Python 2と3のとをたせるには、のように6つのパッケージをします

```
import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('Cannot open {}'), None)
```

イテレータの`.next`メソッドのがされました

Python 2では、イテレータで`next`というメソッドをして、イテレータをトラバースすることができます。

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
```

Python 3では、`.next`メソッドのが`__next__`にされ、"magic"ロールとして`.next`されます。したがって、`.next`をびすと`AttributeError`ます。Python 2とPython 3のでこのにアクセスするしいは、イテレータをとして`next`をびすことです。

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
```

このコードは、2.6からのリリースまでのバージョンでです。

なるタイプの

Python 2.x 2.3

なるタイプのオブジェクトをすることができます。はですがしています。それらは、`None`がよりもなく、がよりもさく、そのすべてがによってにべられるようにけられます。したがって、`int`は`str`よりさく、`tuple`は`list`よりもきい

```
[1, 2] > 'foo'
# Out: False
(1, 2) > 'foo'
# Out: True
[1, 2] > (1, 2)
# Out: False
100 < [1, 'x'] < 'xyz' < (1, 'x')
# Out: True
```

これはもともとは、のリストをソートすることができ、オブジェクトをごとにグループするため
にわれました。

```
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

なるのをするときにかします。

```
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()
```

Python 3でリストをにソートし、バージョンのをするには、ソートされたにキーをするがあります。

```
>>> 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}]
```

`key`として`str`をすると、のでのみかににされます。それから、`[`、`'`、`{` 0-9いずれかでまるを
と、それらのをすべてソートすることができます。

ユーザ

Python 2では、`raw_input`をしてユーザをけれ、

Python 2.x 2.3

```
user_input = raw_input()
```

Python 3では、`input` をしてユーザーをけます。

Python 3.x 3.0

```
user_input = input()
```

Python 2では、`input` はをけてします。これはですが、いくつかのセキュリティのがあり、Python 3ではされました。じにアクセスするには、`eval(input())` をできます。

スクリプトを2つのバージョンでにするには、のコードをPythonスクリプトののくにくことができます。

```
try:
    input = raw_input
except NameError:
    pass
```

ディクショナリの

Python 3では、メソッドのくはPython 2とはかなりっていて、くはされています `has_key`、`iter*`、`view*` はなくなりました。いされていた `d.has_key(key)` わりに `key in d` するがあります。

Python 2では、のメソッドの `keys`、`values`、および `items` はリストをします。Python 3では、わりにビューオブジェクトをします。ビューオブジェクトはイテレータではなく、の2つのでなります。

- らはサイズをとっています1つは `len` をうことができます
- らはもりすことができます

さらに、イテレータとに、ディクショナリのはビューオブジェクトにされます。

Python 2.7は、これらのメソッドをPython 3からバックポートしました。 `viewkeys`、`viewvalues`、`viewkeys`、`viewvalues` としてでき `viewitems`。 Python 2コードをPython 3コードにするには、するフォームをのようにします。

- Python 2の `d.keys()`、`d.values()`、`d.items()` は `list(d.keys())`、`list(d.values())`、`list(d.items())`
- `d.iterkeys()`、`d.itervalues()` および `d.iteritems()` は `iter(d.keys())` にするか、`iter(d)` するがあります。 `iter(d.values())` および `iter(d.items())` それぞれ
- そしてにはPython 2.7メソッド `d.viewkeys()`、`d.viewvalues()` と `d.viewitems()` できえることができる `d.keys()` `d.values()` と `d.items()`

Python 2のコードをすることは、のキー、、またはをしながらそれをするとは々しいことがあります。してください

```
d = {'a': 0, 'b': 1, 'c': 2, '!': 3}
for key in d.keys():
```

```
if key.isalpha():
    del d[key]
```

コードはPython 3でもに`keys`が、`keys`メソッドはリストではなくビューオブジェクトをします。もしにのサイズがされると、Python 3のコードは`RuntimeError: dictionary changed size during iteration`クラッシュし`RuntimeError: dictionary changed size during iteration`。もちろん、は`for key in list(d)`をしきむこと`for key in list(d)`。

に、ビューオブジェクトはイテレータとはなるをします。つまり、`next()`オブジェクトをすることはできず、をすることはできません。わりにします。Python 2のコードが`d.iterkeys()`、`d.itervalues()`または`d.iteritems()`のりを、*iterable*のわりに*iterator*をとするメソッドに`d.iteritems()`、`iter(d)`、`iter(d.values())` Python 3では`iter(d.values())`または`iter(d.items())`します。

execはPython 3のです

Python 2では、`exec`はなをつです `exec code [in globals[, locals]]`. Python 3では、`exec`は`exec(code, [, globals[, locals]])`になり、Python 2のは`SyntaxError`ます。

`print`がstatementからにされたときに、`__future__ import`もされました。しかし、`from __future__ import exec_function`はありません`from __future__ import exec_function` 2の`exec`は、Python 3の`exec`のびしとまったくじですすることもできます。

Python 2.x 2.3

```
exec 'code'
exec 'code' in global_vars
exec 'code' in global_vars, local_vars
```

フォームに

Python 3.x 3.0

```
exec('code')
exec('code', global_vars)
exec('code', global_vars, local_vars)
```

のは、Python 2とPython 3のでじようにすることがされています。

Python 2のhasattrのバグ

Python 2では、プロパティがエラーをさせると、`hasattr`はこのプロパティをして`False`します。

```
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
```

このバグはPython3でされています。したがって、Python 2をするは、

```
try:
    a.get
except AttributeError:
    print("no get property!")
```

わりに`getattr`してください

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("no get property!")
```

モジュールの

ライブラリのいくつかのモジュールのがされました

	しい
<code>_winreg</code>	ウィグレグ
<code>ConfigParser</code>	<code>configparser</code>
<code>copy_reg</code>	コピーレグ
キュー	キュー
<code>SocketServer</code>	ソケットサーバー
<code>_markupbase</code>	マークアップベース
<code>repr</code>	<code>reprlib</code>
<code>test.test_support</code>	<code>test.support</code>
<code>Tkinter</code>	トキンター

	しい
tkFileDialog	tkinter.filedialog
urllib / urllib2	urllib、urllib.parse、urllib.error、urllib.response、urllib.request、urllib.robotparser

のモジュールは、ファイルからライブラリにさえさえあります。としてからtkinterとurllibをとってください。

Python 2.xと3.xののバージョンのをする、[future](#)パッケージをして、Python 2.xバージョンでPython 3.xのをつトップレベルのライブラリパッケージをインポートすることができます。

オクタル

Python 2では、8のリテラルはのようになります。

```
>>> 0755 # only Python 2
```

をするには、

```
0o755 # both Python 2 and Python 3
```

すべてのクラスは、Python 3では「しいスタイルのクラス」です。

Python 3.xすべてのクラスはしいスタイルのクラスです。しいクラスをするとき、Pythonはにobjectからしobject。そのため、classでobjectをするobjectはにオプションです。

Python 3.x 3.0

```
class X: pass
class Y(object): pass
```

これらのクラスはどちらも、objectをmro メソッドにmro object。

Python 3.x 3.0

```
>>> X.__mro__
(__main__.X, object)

>>> Y.__mro__
(__main__.Y, object)
```

Python 2.xクラスは、デフォルトでいスタイルのクラスです。らはのうちにobjectからしません。これにより、classとしてobjectをにするかどうかによって、クラスのセマンティクスがなります。

Python 2.x 2.3

```
class X: pass
class Y(object): pass
```

この、`Y`の`__mro__`をしようとする、Python 3.xとのがされます

Python 2.x 2.3

```
>>> Y.__mro__
(<class '__main__.Y'>, <type 'object'>)
```

これは、に`Y`をオブジェクトからさせるために、`class Y(object): pass`するためにします。オブジェクトからしないクラス`X`、`__mro__`はしません。アクセスしようすると`AttributeError`ます。

のPythonバージョンのをするために、`object`をクラスとしてクラスをすることができます。

```
class mycls(object):
    """I am fully compatible with Python 2/3"""
```

あるいは、`__metaclass__`がグローバルスコープで`type`にされている、`__metaclass__`されたモジュールでされるすべてのクラスは、`object`からにするがなくのうちにしいスタイルになり`object`。

```
__metaclass__ = type

class mycls:
    """I am also fully compatible with Python 2/3"""
```

された`<>`と```は、`=`と`repr`とです。

Python 2では、`<>`は`!=`です。に、``foo``は`repr(foo)`です。

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
```

もはやな16へのエンコード/デコード

Python 2.x 2.7

```
"1deadbeef3".decode('hex')  
# Out: '\x1d\xea\xdb\xee\xfb'  
'\x1d\xea\xdb\xee\xfb'.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\xfb'.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\xfb'.encode('hex')  
# Traceback (most recent call last):  
#   File "<stdin>", line 1, in <module>  
# AttributeError: 'bytes' object has no attribute 'encode'
```

ただし、エラーメッセージでされるように、[codecs](#) モジュールをしてじをすることができます。

```
import codecs  
codecs.decode('1deadbeef4', 'hex')  
# Out: b'\x1d\xea\xdb\xee\xfc'  
codecs.encode(b'\x1d\xea\xdb\xee\xfc', 'hex')  
# Out: b'1deadbeef4'
```

[codecs.encode](#) は `bytes` オブジェクトをします。ASCII に [decode](#) する `str` オブジェクトをするには

```
codecs.encode(b'\x1d\xea\xdb\xee\xff', 'hex').decode('ascii')  
# Out: '1deadbeeff'
```

Python 3でcmpがされました

Python 3では、[cmp](#) みみが `__cmp__` メソッドとともにされました。

ドキュメントから

`cmp()` は `__cmp__()` としてわれ、 `__cmp__()` メソッドはサポートされなくなりました。ソートには `__eq__()`、 `__hash__()` は `__lt__()` をし、にじてのなをします。に `cmp()` がなは、 $(a > b) - (a < b)$ を `cmp(a, b)` としてできます。

さらに、 `cmp` パラメーターをくれたみみはすべて、 `key` キーワードのみのパラメーターをけるようになります。

で `functools` モジュールにもながあり `cmp_to_key(func)` あなたからすることができ `cmp` にスタイルのファンクション `key` スタイルの

いスタイルのをキーにします。キー `sorted()`、 `min()`、 `max()`、 `heapq.nlargest()`、 `heapq.nsmallest()`、 `itertools.groupby()` をけるツールでされます。このはに、のをサポートする Python 2 からされるプログラムのツールとしてされます。

リストのにおけるれ

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!'
```

このからわかるように、Python 2 では `x` のがれました。それは `hello world!` マスクしました `hello world!` これはループがしたときの `x` ののだったので、 `U` します。

しかし、Python 3 `x` では、にされた `hello world!` プリントされ `hello world!` リストのローカルは、のスコープのをマスクしないためです。

さらに、ジェネレータ Python 2.5 でや、やの Python 3 から Python 2.7 にバックポートされたは Python 2 でをリークしません。

Python 2 と Python 3 ので、`for` ループをすると、がのスコープにれることにしてください。

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
```

```
print(x)
# Out: 'U'
```

`map()` は、`iterable` のにをするのになみみです。Python 2では、`map` はリストをします。Python 3では、`map` はジェネレータであるマップオブジェクトをします。

```
# 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']
```

Python 2では、`None` をしてIDとしてさせることができます。これはもはやPython 3ではしません。

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
```

さらに、Python 2でとしての`iterable`をす、`map` はい`iterable` を `None` `itertools.izip_longest` で `itertools.izip_longest` で `itertools.izip_longest` ます。Python 3では、はののにします。

Python 2では

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)]
```

Python 3では

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)]
```

`map` 代わりに、リスト Python 2/3 をすることをしてください。 `map(str, [1, 2, 3, 4, 5])` さえ `map(str, [1, 2, 3, 4, 5])`

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

filter、**map** および **zip** はシーケンスのわりにイテレータをします

Python 2.x 2.7

Python 2 の `filter`、`map` と `zip` みみがシーケンスをします。 `map` と `zip` にリストをしますが、`filter` はりののはえられたパラメータのにします

```
>>> 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

Python 3 `filter`、代わりに `map` と `zip` がイテレータをします。

```
>>> 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)]
```

Python 2 `itertools.izip` は Python 3 とです `zip` `izip` は Python 3 でされています。

Python 3では、[PEP 404](#)はPython 2からのインポートのをします。のインポートはパッケージや `from ... import *` `imports`はモジュールレベルのコードでのみされています。

Python 2でPython 3のをするには

- [インポート](#)は `from __future__ import absolute_import`にすることができます `from __future__ import absolute_import`
- なは、ののわりにされる

にするために、Python 2では、モジュールはののようにじディレクトリにあるのモジュールのをインポートできます

```
import foo
```

`foo`のは、`import`だけではあいまいであることにしてください。このようなのインポートは、[なインポート](#)をしています。のようになります。

```
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
```

ドット.ディレクトリツリーのモジュールをにできます。

の

`shapes`とばれるユーザーのパッケージについてえてみましょう。ディレクトリはのとおりで。

```
shapes
├── __init__.py
│
├── circle.py
│
├── square.py
│
└── triangle.py
```

`circle.py`、`square.py`、および `triangle.py` すべてモジュールとして `util.py` をインポートします。じレベルのモジュールをどのようにするのでしょ

```
from . import util # use util.PI, util.sq(x), etc
```


または

```
from .util import * #use PI, sq(x), etc to call functions
```

.じレベルのなインポートにされます。

に、`shapes`モジュールのレイアウトをえてみましょう。

```
shapes
├── __init__.py
│
├── circle
│   ├── __init__.py
│   └── circle.py
│
├── square
│   ├── __init__.py
│   └── square.py
│
├── triangle
│   ├── __init__.py
│   └── triangle.py
│
└── util.py
```

さて、これらの3つのクラスは`util.py`をどのようにしますか

```
from .. import util # use util.PI, util.sq(x), etc
```

または

```
from ..util import * # use PI, sq(x), etc to call functions
```

`..`は、レベルのインポートにされ`..`。もっとしてください`.`とのレベルのを。

ファイルI/O

`file`はもはや`3.x`のみものではありませんまだ`open`います。

ファイルI/Oののは、ライブラリ`io`モジュールにされました。これは`StringIO`しいホームです。

```
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'
```

ファイルモードテキストバイナリは、ファイルを読み込むことによってされるデータのタイプおよびきみになタイプをするようになりました。

```
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
```

テキストファイルのエンコーディングは、デフォルトで `locale.getpreferredencoding(False)` によってされるものになります。エンコードをにするには、 `encoding` キーワードパラメータをします。

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

round のタイプブレイクとりの

ラウンドタイプブレイク

Python 2では、2つのにいくいに `round()` をすると、0からもしものものがされます。

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
```

しかし、Python 3では、 `round()` はのをしますのめ。えは

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
```

ラウンドは、のにもめにそのもいのになります、ラウンドのえは、 `round(2.5)` 2なく、3.0をします。

[Wikipediaのによると](#)、これはめ、め、のめ、オランダのめ、ガウスめ、または - めともばれます。

からへのめは [IEEE 754](#) のであり、Microsoft の .NET のデフォルトのめモードでもあります。

このめは、めのをらすがあります。められたのはしてりてられたのと同じなので、めはキャンセルされます。わりに、のめは、にまたはバイアスをするがある。

round りの

`round()` は、Python 2.7で `float` をします

Python 2.x 2.7

```
round(4.8)
# 5.0
```

Python 3.0から、2のをすると、 `int` します。

Python 3.x 3.0

```
round(4.8)
# 5
```

、なし

Python 2では `True`、`False`、`None` はみみです。つまり、それらをりてすることができます。

Python 2.x 2.0

```
True, False = False, True
True # False
False # True
```

Python 2.4、これを `None` うことはできません。

Python 2.x 2.4

```
None = None # SyntaxError: cannot assign to None
```

Python 3では、`True`、`False`、`None` はキーワードになりました。

Python 3.x 3.0

```
True, False = False, True # SyntaxError: can't assign to keyword
None = None # SyntaxError: can't assign to keyword
```

ファイルオブジェクトへのきみのり

Python 2では、ファイルハンドルにきむと `None` がされます。

Python 2.x 2.3

```
hi = sys.stdout.write('hello world\n')
# Out: hello world
type(hi)
# Out: <type 'NoneType'>
```

Python 3では、ハンドルにきむと、テキストをきむときにきまれると、バイトをきむときにきま

れるバイトがされます。

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
```

いint

Python 2では、C `ssize_t`よりきいのが、リテラルに`L`でされる`long`データにされます。たとえば、Pythonの32ビットビルドでは、のようになります。

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
```

しかし、Python 3では、`long`データがされました。がどれほどきくても`int`ます。

Python 3.x 3.0

```
2**1024
# Output:
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021...

print(-(2**1024))
# Output: -
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021...

type(2**1024)
# Output: <class 'int'>
```

クラスブール

Python 2.x 2.7

Python 2では、クラスのブールをでするは、クラスに`__nonzero__`メソッドをするがあります。は

デフォルトでTrueです。

```
class MyClass:
    def __nonzero__(self):
        return False

my_instance = MyClass()
print bool(MyClass)      # True
print bool(my_instance)  # False
```

Python 3.x 3.0

Python 3では、__bool__わりに__nonzero__がされています

```
class MyClass:
    def __bool__(self):
        return False

my_instance = MyClass()
print (bool(MyClass))    # True
print (bool(my_instance)) # False
```

オンラインでPython 2からPython 3へのをむ <https://riptutorial.com/ja/python/topic/809/python-2からpython-3への>

37: Python HTTP Server

Examples

シンプルなHTTPサーバーの

Python 2.x 2.3

```
python -m SimpleHTTPServer 9000
```

Python 3.x 3.0

```
python -m http.server 9000
```

このコマンドをすると、のディレクトリのファイルがポート 9000 でされます。

ポートとしてがされていない、サーバーはデフォルトポート 8000 されます。

-m フラグは sys.path をし、する .py ファイルがモジュールとしてされるようにします。

あなたがlocalhostでのみしたいなら、のようなカスタムPythonプログラムをくがあります

```
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()
```

ファイルの

のファイルのディレクトリがあるとします。



のようにこれらのファイルをするようにWebサーバーをできます。

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()
```

`SocketServer` モジュールは、ネットワークサーバーをセットアップするためのクラスとをします。

`SocketServer` の `TCPServer` クラスは、TCP プロトコルをしてサーバーをします。コンストラクタはサーバーのアドレスすなわち IP アドレスとポートとサーバーをするクラスをすタプルをけります。

`SimpleHTTPServer` モジュールの `SimpleHTTPRequestHandler` クラスをすると、のディレクトリのファイルをできます。

スクリプトをじディレクトリにしてします。

HTTP サーバーをします。

Python 2.x 2.3

```
python -m SimpleHTTPServer 8000
```

Python 3.x 3.0

```
python -m http.server 8000
```

'-m'フラグは 'sys.path'をし、する '.py'ファイルがモジュールとしてされるようにします。

ブラウザーで[localhost8000](http://localhost:8000)をくと、のようになります。

Directory listing for /

- [facade.py](#)
- [factory.py](#)
- [server.py](#)

SimpleHTTPServerのプログラミングAPI

`python -m SimpleHTTPServer 9000`をするとどうなりますか

このにえるために、SimpleHTTPServer

<https://hg.python.org/cpython/file/2.7/Lib/SimpleHTTPServer.py>とBaseHTTPServer

<https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py> のをするがあります。

に、Pythonは9000をとしてSimpleHTTPServerモジュールをびします。 SimpleHTTPServerコードを
て、

```
def test (HandlerClass = SimpleHTTPRequestHandler,
          ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test (HandlerClass, ServerClass)

if __name__ == '__main__':
    test ()
```

テストは、ハンドラとServerClassのにびされます。すぐBaseHTTPServer.testがびされました

```
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()
```

したがってここでは、ユーザーがとしてしたポートがされ、ホストアドレスにバインドされます。されたポートとプロトコルによるソケットプログラミングのさらになステップがされます。にソケットサーバーがします。

これは、SocketServerクラスからのクラスへののなです。

```
+-----+
| BaseServer |
+-----+
      |
      v
+-----+ +-----+
| TCPServer |----->| UnixStreamServer |
+-----+ +-----+
      |
      v
+-----+ +-----+
| UDPServer |----->| UnixDatagramServer |
+-----+ +-----+
```

リンク<https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py>と
<https://hg.python.org/cpython/file/2.7/Lib/SocketServer.py>は、さらにするのにです。

BaseHTTPRequestHandlerをしたGET、POST、PUTのな

```
# 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()
```

curlをした

```
$ 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
```

オンラインでPython HTTP Serverをむ <https://riptutorial.com/ja/python/topic/4247/python-http-server>

38: Python Lex-Yacc

き

PLYは、なコンパイラツールlexとyaccのなPythonです。

そのリンク

1. [ドキュメント](#)
2. [ギター](#)

Examples

PLY

Python2 / 3のマシンにPLYをインストールするには、のってください

1. [ここ](#)からソースコードをダウンロードして[ください](#)。
2. ダウンロードしたzipファイルをする
3. したply-3.10フォルダにします。
4. でのコマンドをします `python setup.py install`

のすべてをしたら、PLYモジュールをできるようになります。Pythonインタプリタをき、`import ply.lex`としてテストすることができます。

PLYをインストールするためにpipをしないでください。マシンにれたディストリビューションがインストールされます。

「こんにちは、」の**PLY** - な

なをげてPLYのをしましょう。このプログラムはをとしてけり、それをしようします。

きなエディタをき、のコードをコピーしてください

```
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)
```

このファイルを `calc.py` としてしてください。

-8

`-4 * - (3 - 5)` はどちらですか

パート1 **Lex** でトークンする

つは、されたトークンをし、をすることをむれ、それはをするシンボルをし、そしてのステップは、されたと、 トークンされた1からのコードをうことを2つのステップがあります。

このでは、ユーザーをトークンして1ずつかくするのなをします。

```
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)

```

このファイルを `calcllex.py` としてします。Yaccパーサーをビルドするときにこれをします。

す

1. `import ply.lex` をしてモジュールを `import ply.lex`
2. すべてのレクサーは、レクサーができるすべてのなトークンを `tokens` とばれるリストをすることがあります。このリストはにです。

```

tokens = [
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
]

```

`tokens` は、ではなくのタプルでもあり、はとじようにトークンをします。

3. のルールは、またはとしてできます。いずれのも、のに `t_` がいていなければなりません。
 - なトークンの、はとしてできます `t_PLUS = r'\+'`
 - らかのアクションをするがある、トークンルールをとしてできます。

```

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

```

ルールはのドキュメントとしてされていることにしてください。これは、`LexToken`インスタンスである1つのを`LexToken`、らかのアクションをしてからをします。

ドキュメントをするわりに、のルールとしてをするは、のをしてください。

```
@TOKEN(identifier)          # identifier is a string holding the regex
def t_ID(t):
    ...                      # actions
```

- `LexToken`オブジェクトのインスタンスこのオブジェクトを`t`とぶには、のがあります。

1. `t.type`はトークンタイプです `'NUMBER'`、`'PLUS'`など。デフォルトでは、`t.type`は`t_`ののにされます。
2. `t.value`ある`t.value`のテキストにマッチしたもの
3. のである`t.lineno`これはにはされません。これは、レクサーがをもらないためです。`t_newline`というをとって`lineno`をして`t_newline`。

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

4. `t.lexpos`は、テキストのをとするトークンのです。

- `regex`ルールからもされなければ、トークンはされます。トークンをしたいは、`t_ignore_`をじルールのをするのではなく、のルールにすることもできます。

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # No return value. Token discarded
```

...じです

```
t_ignore_COMMENT = r'\#.*'
```

これは、あなたがコメントをたときにらかのアクションをしている、もちろんです。その、ルールをするをします。

いくつかのにしてトークンをしていないがそれでもしたいは、`t_ignore =`
"`<characters to ignore>`" これらのはです。

```
t_ignore_COMMENT = r'\#.*'
t_ignore = ' \t'      # ignores spaces and tabs
```

- マスターをするとき、`lex`はファイルにされたをのようになります

1. でされたトークンは、ファイルにされるとじでされます。

2. でされたトークンは、そのトークンのをすののにされます。

`==`と`=`をじファイルにマッチさせるは、これらのをしてください。

- リテラルはそのままされるトークンです。 `t.type`と`t.value`がそのものにされます。 リテラルのリストをのようになります。

```
literals = [ '+', '-', '*', '/' ]
```

または、

```
literals = "+-*/"
```

リテラルがしたときにアクションをするトークンをすることはです。ただし、トークンのをにするがあります。えは

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = '{' # Set token type to the expected literal (ABSOLUTE MUST if this
                  # is a literal)
    return t
```

- `t_error`でエラーをします。

```
# 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)
```

に、 `t.lexer.skip(n)` はの`n`のをスキップします。

4. な

`lexer = lex.lex()` をしてレクサーをします。

また、クラスのすべてをし、クラスのインスタンスをしてレクサーをすることもできます。
えは

```
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")    #

```

データがであるlexer.input(data)をしてをする

トークンをするには、したトークンをするlexer.token()をします。のようにループでレクサーをできます。

```

for i in lexer:
    print(i)

```

パート2トークンされたをYaccでする

このセクションでは、パート1のトークンされたがどのようにされるかについてします。これは、コンテキストフリーグラマーCFGをしてわれま。をしなればならず、トークンはにってされま。フードのでは、パーサーはLALRパーサーをしま。

```

# 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)

```

す

- は、そのにするdocstringがなをむによってされる。をするステートメントは、ルールのセマティックアクションをします。は、するルールののをむシーケンスであるのpをけります
 - p[i]のは、のようにシンボルにマッピングされます。

```

def p_expression_plus(p):
    'expression : expression PLUS term'
    #      ^           ^           ^   ^
    # p[0]         p[1]         p[2] p[3]

    p[0] = p[1] + p[3]

```

- トークンの、するp[i]の""は、レクサーモジュールで与えられたp.valueと同じです。したがって、PLUSは+をちます。
- の、はp[0]されているものによってまります。もされていない、はNoneです。また、p[-1]と同じではないp[3]ので、pなりリストないp[-1]めまれたアクションここではしないをすることができます。

このは、p_がにいてるり、のをつことができます。

- p_error(p) ルールはエラーをするようにされていますyacc / bisonのyyerrorと同じ。
- のルールを1つのにまとめることができます。これは、プロダクションのがしていればいえます。

```
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]
```

- トークンのわりにリテラルをできます。

```
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]
```

もちろん、リテラルはレクサーモジュールですがあります。

- のプロダクションは、`'''symbol : '''`ついでに`'''symbol : '''`
- にシンボルをするには、`start = 'foo'`します `foo`は`start = 'foo'`です。
- precedenceとassociativityのは、precedenceをしようことができます。

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'), # Nonassociative operators
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'), # Unary minus operator
)
```

トークンのは、のいにべられます。 `nonassoc`は、それらのトークンがしないことをします。これは、`a < b < c`ようなものはであり`a < b`はとしてであることをする。

- `parser.out`は、yaccプログラムがめてされたときにされるデバッグファイルです。シフト/リダクションのがするたびに、パーサーはにシフトします。

オンラインでPython Lex-Yaccをむ <https://riptutorial.com/ja/python/topic/10510/python-lex-yacc>

39: Python Server Sent イベント

き

Server Sent EventsSSEは、サーバーとクライアントはWebブラウザーなので、サーバーがクライアントに「プッシュ」できるようにします。これはウェブソケットやロングポーリングによくなっています。SSEとWebソケットのなには、SSEはであり、サーバーだけがクライアントにをすることができます。WSocketのとな、ともにをできます。SSEは、ウェブソケットよりもはるかにに/できるとえられています。

Examples

フラスコ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

ここでは、asyncio SSEライブラリをしています。 <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()
```

オンラインでPython Server Sentイベントをむ <https://riptutorial.com/ja/python/topic/9100/python-server-sentイベント>

40: Python シリアル pyserial

- `ser.read` サイズ = 1
- `ser.readline`
- `ser.write`

パラメーター

パラメーター	
	デバイス GNU / Linux では / dev / ttyUSB0、Windows では COM3。
ボーレート	ボーレートタイプ int デフォルト 9600 50,75,110,134,150,200,300,600,1200,1800,2400,4800,9600,19200,38400,57600,115200

については、 [pyserial documentation](#) をしてください。

Examples

シリアルデバイスをする

```
import serial
#Serial takes these two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

シリアルポートからのみり

シリアルデバイスをする

```
import serial
#Serial takes two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

シリアルデバイスからシングルバイトをみる

```
data = ser.read()
```

シリアルデバイスからえられたバイトをみむ

```
data = ser.read(size=5)
```

シリアルデバイスから1をみみます。

```
data = ser.readline()
```

かがきされているにシリアルデバイスからデータをみみます。

```
#for python2.7
data = ser.read(ser.inWaiting())

#for python3
ser.read(ser.inWaiting)
```

マシンでなシリアルポートをする

なシリアルポートのリストをするには

```
python -m serial.tools.list_ports
```

コマンドプロンプトまたは

```
from serial.tools import list_ports
list_ports.comports() # Outputs list of available serial ports
```

Pythonシェルから。

オンラインでPythonシリアルpyserialをむ <https://riptutorial.com/ja/python/topic/5744/pythonシリアル-pyserial->

41: Python データ

き

データはメモリのスペースを確保するために使われます。Pythonは、メモリを確保するためのものをとれません。は、にを確保するとに変わります。

Examples

データ

には、Pythonでは4つのタイプがあります。int、float、complex、およびlongです。

```
int_num = 10      #int value
float_num = 10.2   #float value
complex_num = 3.14j #complex value
long_num = 1234567L #long value
```

データ

は、で囲まれた文字列として扱われます。Pythonでは、またはのいずれかをすることができます。はのシーケンスデータです。つまり、を確保するたびに、新しいオブジェクトが作成されます。

```
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
```

リストデータタイプ

リストにはコンマで区別され、[]で囲まれたデータで構成されています。リストはCの配列とほぼ同じです。1つの要素は、リストに属するすべてのデータであるということです。

```
list = [123,'abcd',10.2,'d']    #can be a array of any data type or single data type.
list1 = ['hello','world']
print(list)      #will output whole list. [123,'abcd',10.2,'d']
print(list[0:2]) #will output first two element of list. [123,'abcd']
print(list1 * 2)  #will give list1 two times. ['hello','world','hello','world']
print(list + list1) #will give concatenation of both the lists.
[123,'abcd',10.2,'d','hello','world']
```

タプルデータ

リストはカッコ[]で囲まれており、そのサイズは変更できますが、タプルはカッコで囲まれており、変更できません。タプルはです。

```

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.

```

データ

ディクショナリはキーとのペアでされています。{}でまれ、[]をしてをりててアクセスできます。

```

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']

```

データの

はのオブジェクトのものであり、には2つのタイプがある

1. セット - それらはであり、セットがされるとしいエレメントをできます

```

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. セット - であり、されたでしいエレメントをすることはできません。

```

b = frozenset('asdfagsa')
print(b)
> frozenset({'f', 'g', 'd', 'a', 's'})
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
print(cities)
> frozenset({'Frankfurt', 'Basel', 'Freiburg'})

```

オンラインでPythonデータをむ <https://riptutorial.com/ja/python/topic/9366/pythonデータ>

42: Pythonでのセキュアシエル

パラメーター

パラメータ	
ホスト	このパラメータは、をやるがあるホストをします
ユーザー	ホストにアクセスするためになユーザー
ホストポート	
パスワード	アカウントのパスワード

Examples

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
```

オンラインでPythonでのセキュアシエルをむ <https://riptutorial.com/ja/python/topic/5709/pythonでのセキュアシエル>

43: Python でのない

Cursesは、Pythonのなまたはモジュールです。これは、ターミナルベースのユーザーインターフェイスまたはTUIをするためにできます。

これは、よりなCライブラリ 'ncurses'のPythonポートです

Examples

なびしの

```
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()
```

wrapperヘルパー。

のなびしはですが、cursesパッケージはwrapper(func, ...)ヘルパーをしwrapper(func, ...)。のは、とじものをんでいます

```
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)
```

ここで、ラッパは `curses` をし、 `stdscr`、 `WindowObject` をし、 `stdscr` と `func` それのをします。
`func` がってくると、 `wrapper` はプログラムがするにをします。

オンラインでPythonでのないをむ <https://riptutorial.com/ja/python/topic/5851/pythonでのない>

44: Pythonでのプログラミング

き

プログラミングは、をのにします。には、はのみをけり、をするだけで、えられたにしてされるにをえるはありません。は、ラムダ、マップ、リダクションなどのくにするテクニックです。

Examples

ラムダ

ラムダでされたのインライン。ラムダのパラメータは、コロンのにされています。はコロンのにされています。のはにされます。

```
s=lambda x:x*x
s(2)    =>4
```

Mapはとアイテムのコレクションをります。しいのコレクションをし、のコレクションのアイテムにしてをし、コレクションのりをしいコレクションにします。しいコレクションをします。

これは、のリストをり、それらののさのリストをすなマップです。

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)    =>[4, 4, 3]
```

をらす

Reduceはとアイテムのコレクションをります。アイテムをしてされたをします。

これはなです。コレクションのすべてのアイテムのをします。

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)    =>10
```

フィルター

Filterはとコレクションをります。それは、がTrueをすすべてのアイテムのコレクションをします。

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]    # outputs[5,6]
```

オンラインでPythonでのプログラミングをむ <https://riptutorial.com/ja/python/topic/9552/python>で

45: Python と Excel

Examples

リストのデータを**Excel**のファイルにれます。

```
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はメモリのxlsx/xlsm/xltx/xltmワークブックをおよびするためのモジュールです。

のワークブックをしむ

```
import openpyxl as opx
#To change an existing workbook we located it by referencing its path
workbook = opx.load_workbook(workbook_path)
```

`load_workbook()`にはパラメータ `read_only` がまれています。これを `True` にすると、ブックを `read_only` としてみます。これは、きな `xlsx` ファイルをむときにちます。

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

あなたはメモリにワークブックをロードしたら、して々のシートにアクセスすることができます
`workbook.sheets`

```
first_sheet = workbook.worksheets[0]
```

なシートのをするは、 `workbook.get_sheet_names()` できます。

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

に、 `sheet.rows` をしてシートのにアクセスできます。シートのをするには、のようにします。

```
for row in sheet.rows:
    print row[0].value
```

、 `row` の `rows` のリストである `Cell`、 `S Cell.value` セルのをします。

メモリにしいブックをする

```
#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
```

いくつかのタブのプロパティは `openpyxl` によってできます。たとえば、 `tabColor`

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

したワークブックをするには、

```
wb.save('filename.xlsx')
```

xlsxwriter をして **Excel** チャートをする

```
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 = xlswriter.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)

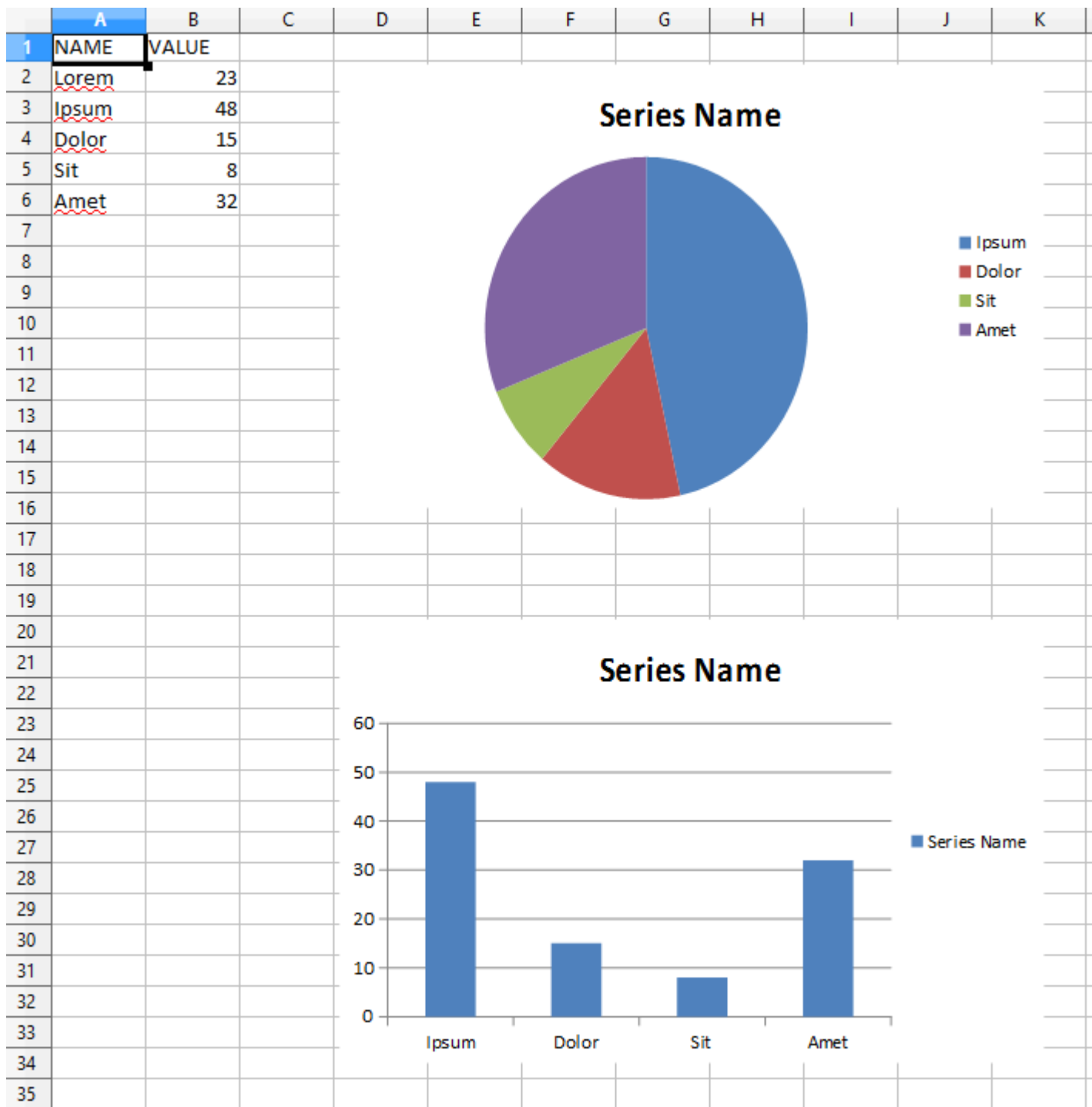
# create column chart
column_chart = workbook.add_chart({'type': 'column'})

# 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()

```

xlrdモジュールをしてExcelデータをむ

Python xlrdライブラリは、Microsoft Excel™スプレッドシートファイルからデータをするこす。

インストール .

```
pip install xlrd
```

または、pypiからsetup.pyファイルをするこもできます

<https://pypi.python.org/pypi/xlrd>

Excelシートをむ - `open_workbook`メソッドをして、`xlrd`モジュールをインポートし、`excel`ファイルを読み込みます。

```
import xlrd
book=xlrd.open_workbook('sample.xlsx')
```

Excelのシートをする

```
print book.nsheets
```

シートをする

```
print book.sheet_names()
```

インデックスについてシートをする

```
sheet=book.sheet_by_index(1)
```

セルのをむ

```
cell = sheet.cell(row,col) #where row=row number and col=column number
print cell.value #to print the cell contents
```

Excelシートのとをする

```
num_rows=sheet.nrows
num_col=sheet.ncols
```

でシートをる

```
sheets = book.sheet_names()
cur_sheet = book.sheet_by_name(sheets[0])
```

Excelファイルを**xlsxwriter**でフォーマットする

```
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()
```

オンラインでPythonとExcelをむ <https://riptutorial.com/ja/python/topic/2986/pythonとexcel>

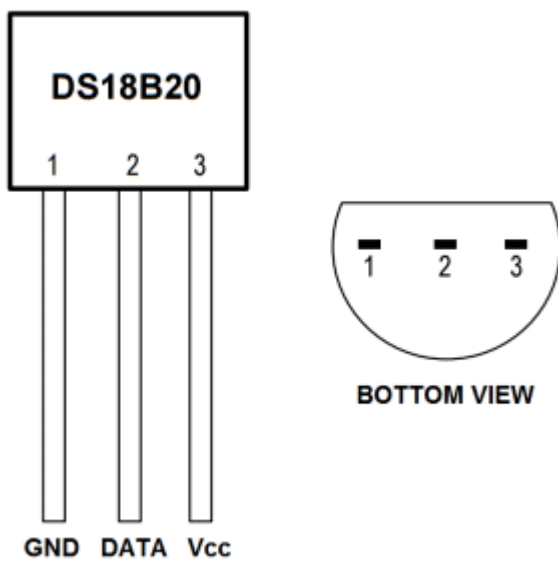
46: PythonとRaspberry PIをしたIoTプログラミング

Examples

・センサー

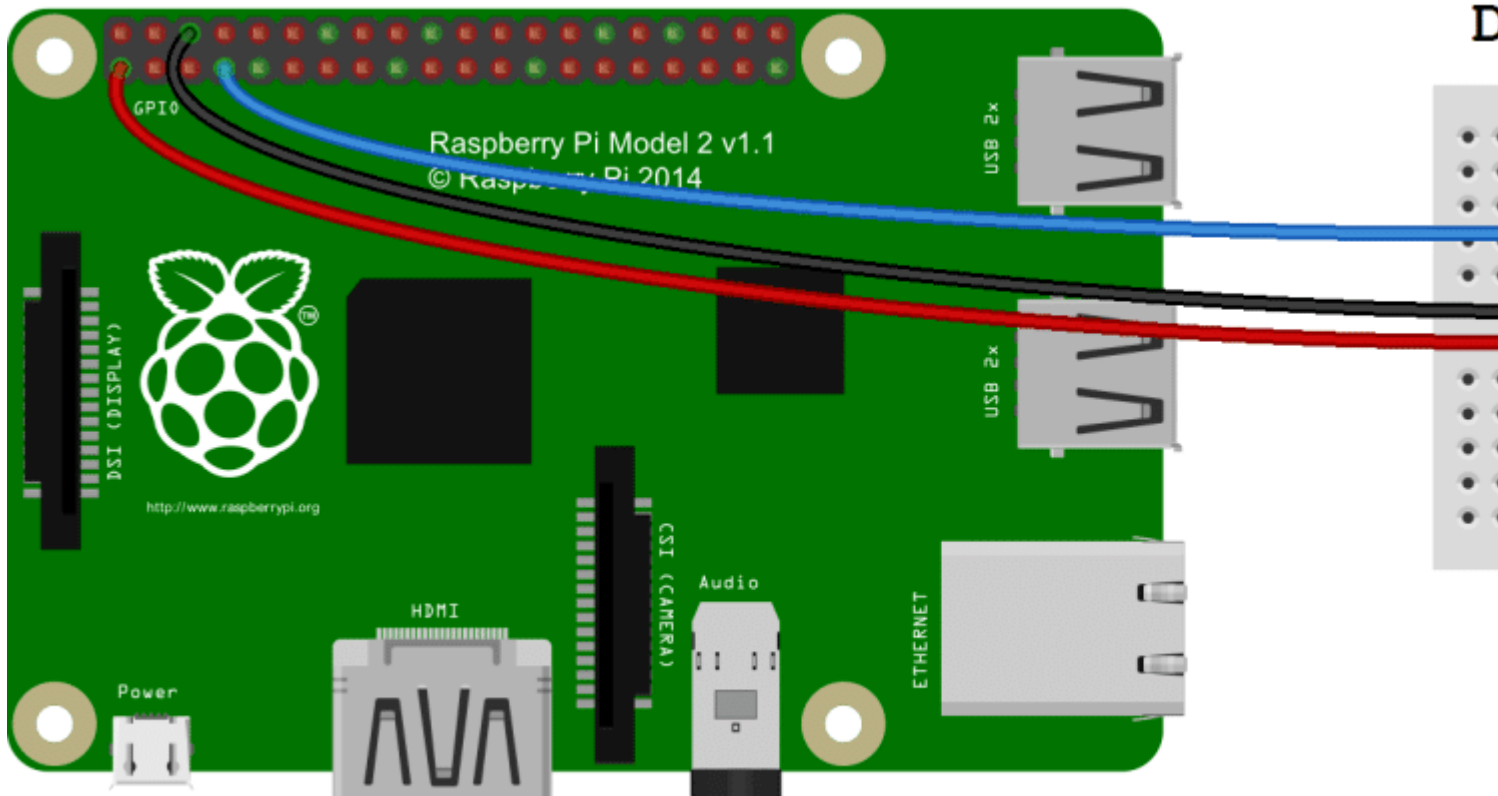
DS18B20とラズベリーパイとのインターフェース

DS18B20とラズベリーパイとの



あなたは3つのターミナルがあることがわかります

1. Vcc
2. Gnd
3. データ1ワイヤプロトコル



R1はレベルをきげるための**4.7k**オームです

1. **Vcc**は、ラズベリーpiPIN01,02,04,17の5vまたは3.3vピンのいずれかにするがあります。
2. **Gnd**はRaspberry piPIN06、09、14、20、25のGndピンのいずれかにするがあります。
3. **DATA**はPIN07にするがあります。

RPiからのインタフェースをにする

4. パテやそののlinux / unixをしてRaspberry piにログインします。
5. ログイン、おにりのブラウザで/boot/config.txtファイルをします。

```
nano /boot/config.txt
```

6. に、この`dtoverlay=w1-gpio`ファイルのにします。
7. は、Raspberry pi `sudo reboot`します。
8. Raspberry piにログインし、 `sudo modprobe g1-gpio`をします。
9. その、 `sudo modprobe w1-therm`します。
10. に、 `/sys / bus / w1 / devices`ディレクトリにします `cd /sys/bus/w1/devices`
11. これで、28 - *****からまるあなたのセンサーでされたディレクトリがわかりました。

12. このディレクトリにく `cd 28-*****`

13. 、 **w1-slave**というファイルがあります。このファイルには、CRCのようなとそののがまれています。 `cat w1-slave`。

Pythonでモジュールをいてをみます

```
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)
```

のpythonモジュールはののとアドレスをします。 RATEパラメータは、センサからのクエリのをまたはするようにされています。

GPIOピン

1. [https://www.element14.com/community/servlet/JiveServlet/previewBody/73950-102-11-339300/pi3_gpio.png] [3]

オンラインでPythonとRaspberry PIをしたIoTプログラミングをむ

<https://riptutorial.com/ja/python/topic/10735/pythonとraspberry-piをしたiotプログラミング>

47: PythonとSQL Serverの

Examples

サーバーに、テーブルの、クエリデータ

パッケージをインストールします。

```
$ 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()
```

あなたのSQLにしているなら、でもできます。ただ、このexecuteメソッドCRUDにすだけです。

withステートメント、ストアドプロシージャのびし、エラーなどのサンプルチェック pymssql.org

オンラインでPythonとSQL Serverのをむ <https://riptutorial.com/ja/python/topic/7985/pythonとsql-serverの>

48: PythonによるWebスクレイピング

き

Webスクレイピングはされたプログラムなプロセスで、データをウェブページからえず「きる」ことができます。スクリーンスクレイピングまたはウェブともばれるウェブスクレイピングは、にアクセスなウェブページからインスタントデータをすることができます。のウェブサイトでは、ウェブのスクレイピングがかもしれません。

WebスクレイピングアルファベットにつPythonパッケージ

リクエストのとデータの

`requests`

HTTPリクエストをするためのなパッケージです。

`requests-cache`

`requests` キャッシュ。キャッシュデータはにです。は、にサイトにヒットすることをけることができます。のコレクションをしているに、あなたのスクレーパーがらかのでクラッシュしたおそらく、サイトのしいコンテンツをしていないかもしれません...おそらくサイトがダウンしたのでしょうか、コレクションをすばやくりすことができますあなたがしたところから。

`scrapy`

Webクローラーをするのにです。 `requests` をしてページをするよりもなものです。

`selenium`

ブラウザのためのSelenium WebDriverのPythonバインディング。 `requests` をしてHTTPリクエストをすることは、ウェブページをするためにしばしばです。しかし、 `requests` でしてサイトのましいるいをすることができない、にJavaScriptがページのをレンダリングするがあるは、これはなツールです。

HTML

`BeautifulSoup`

さまざまなパーサーPythonのみみHTMLパーサ、 `html5lib`、 `lxml`または `lxml.html` をして、HTML

およびXMLドキュメントをクエリします。

[lxml](#)

HTMLとXMLをします。CSSセレクタとXPathをして、HTMLドキュメントからコンテンツをクエリしてするためにできます。

Examples

リクエストとlxmlを使ってデータをスクラップするな

```
# 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()
```

リクエストによるWebスクレイピングセッションの

Cookieやその他のパラメータをするためにWebスクレイプセッションをすることをおめします。さらに、requests.Sessionは、TCPをホストにするため、パフォーマンスがします。

```
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)
```

Scrapyフレームワークをしたスクレイピング

まず、しいScrapyプロジェクトをセットアップするがあります。コードをするディレクトリをし

てします。

```
scrapy startproject projectName
```

スクラップするには、クモがです。スパイダーは、のサイトをどのようにりるかをしします。
StackOverflowでされたへのリンクをたどり、ページ [ソース](#) からいくつかのデータをスクラップするスパイダーのコードはのとおりです。

```
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,
        }
```

spiderクラスをprojectName\spidersディレクトリにします。この、
projectName\spiders\stackoverflow_spider.py。

あなたはあなたのクモをうことができます。たとえば、プロジェクトのディレクトリで試してみ
てください

```
scrapy crawl stackoverflow
```

Scrapyユーザーエージェントをする

デフォルトのScrapyユーザーエージェント "Scrapy/VERSION (+http://scrapy.org)" がホストによっ
てブロックされることがあります。デフォルトのユーザエージェントをするには、 **settings.py**を
き、のコメントをしてします。

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

えば

```
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'
```

BeautifulSoup4をしたスクレイピング

```
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
```

Selenium WebDriverをしたスクレイピング

いくつかのウェブサイトはきられたくありません。このような、ブラウザをとっているのユーザをシミュレートするがあります。SeleniumはWebブラウザをします。

```
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)
```

セレンははるかにくをうことができます。ブラウザのCookieをしたり、フォームにしたり、マウスのクリックをシミュレートしたり、Webページのスクリーンショットをったり、カスタムJavaScriptをしたりすることができます。

urllib.requestによるシンプルなWebコンテンツのダウンロード

ライブラリモジュールurllib.requestをしてWebコンテンツをダウンロードできます

```
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)
```

のモジュールは、[Python 2](#)でもできます。

カールでく

```
from subprocess import Popen, PIPE
from lxml import etree
from io import StringIO
```

ダウンロード

```
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 サイレントダウンロード

-A ユーザエージェントフラグ

する

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

オンラインでPythonによるWebスクレイピングをむ

<https://riptutorial.com/ja/python/topic/1792/pythonによるwebスクレイピング>

49: Pythonによるデータ

Examples

Matplotlib

[Matplotlib](#)は、さまざまなプロットをするPythonのプロットライブラリです。

matplotlibのドキュメントは[ここ](#)にあり、SO Docsは[こちら](#)からできます。

Matplotlibはプロットのための2つの異なるメソッドをしていますが、ほとんどの、それらはです

- まず、matplotlibは、なグラフをMATLABライクなスタイルでプロットすることができる `pyplot` インターフェイス、で使いやすいインターフェイスをします。
- 2に、matplotlibをすると、オブジェクトベースのシステムをして、さまざまなアスペクト、ティックなどをできます。これはよりですが、プロットをにすることができます。

は、されたデータをプロットするために `pyplot` インターフェイスをします

```
import matplotlib.pyplot as plt

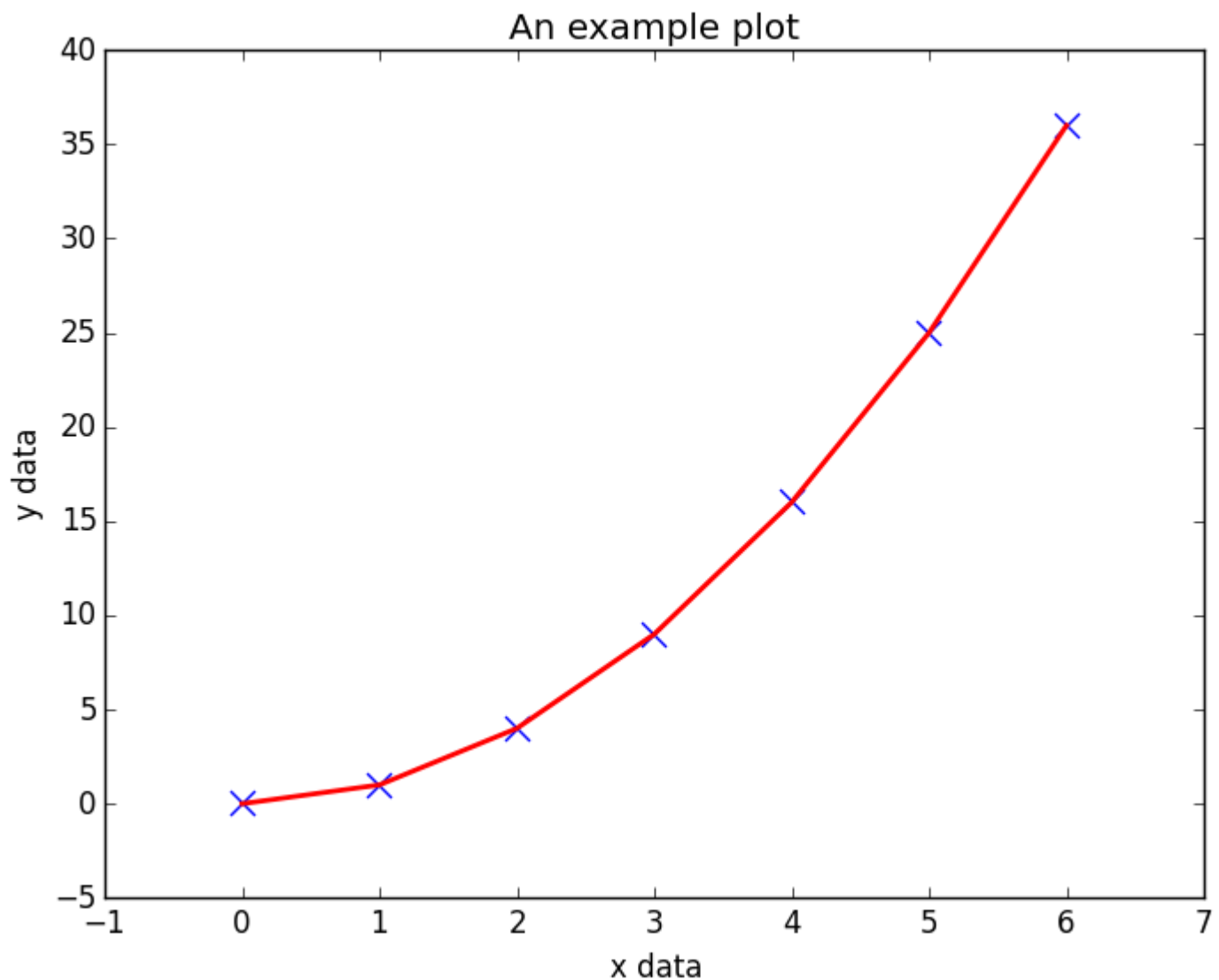
# Generate some data for plotting.
x = [0, 1, 2, 3, 4, 5, 6]
y = [i**2 for i in x]

# 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()
```



モードで`matplotlib.pyplot`をしているため、ので`plt.show()`がになることがあることにしてください。そうであれば、オプションの`plt.show(block=True)`をすることで、ブロックをにオーバーライドできます`plt.show(block=True)`、をする。

シーボーン

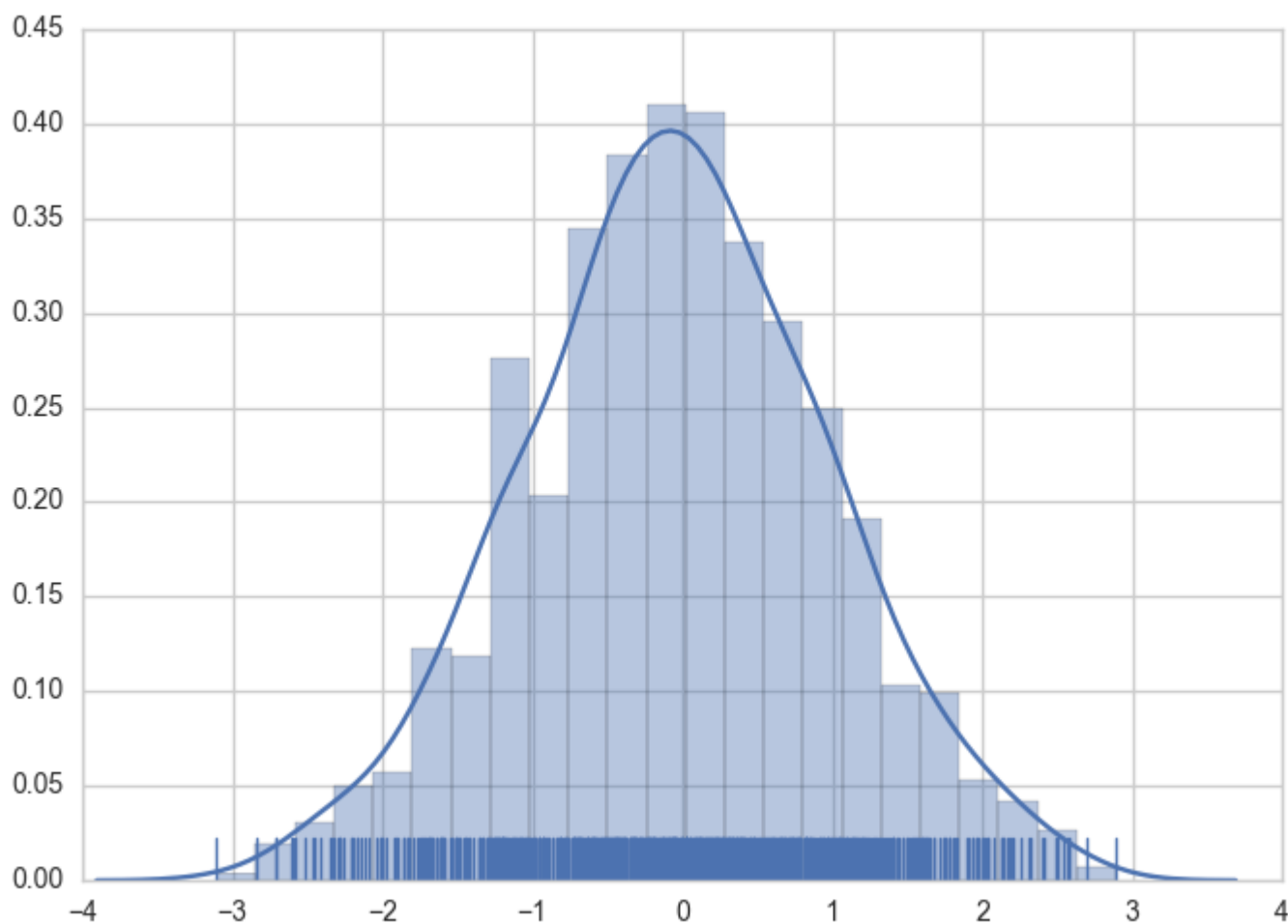
[Seaborn](#)はMatplotlibのラッパーであり、なプロットをにできます。サポートされるプロットのリストには、およびプロット、プロット、およびカテゴリをプロットするためのくのがまれます。Seabornがするプロットのなリストは[APIリファレンス](#)にあります。

Seabornでグラフをするのは、なグラフをひすだけです。に、ランダムにされたデータのヒストグラム、カーネル、およびラグプロットのをします。

```
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)
```

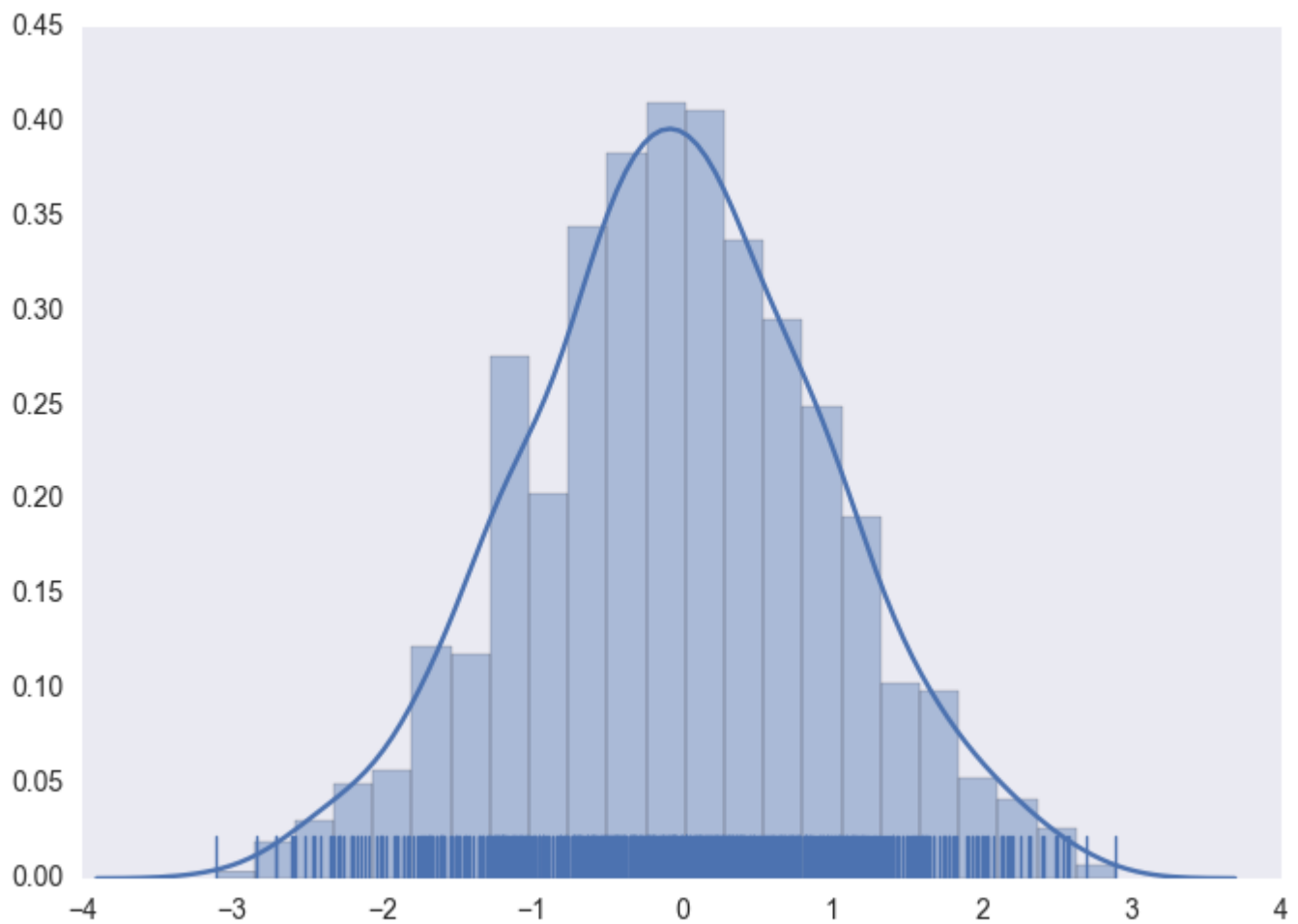
```
# Plot a histogram with both a rugplot and kde graph superimposed
sns.distplot(data, kde=True, rug=True)
```



プロットのスタイルは、をしてすることもできます。

```
# 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)
```

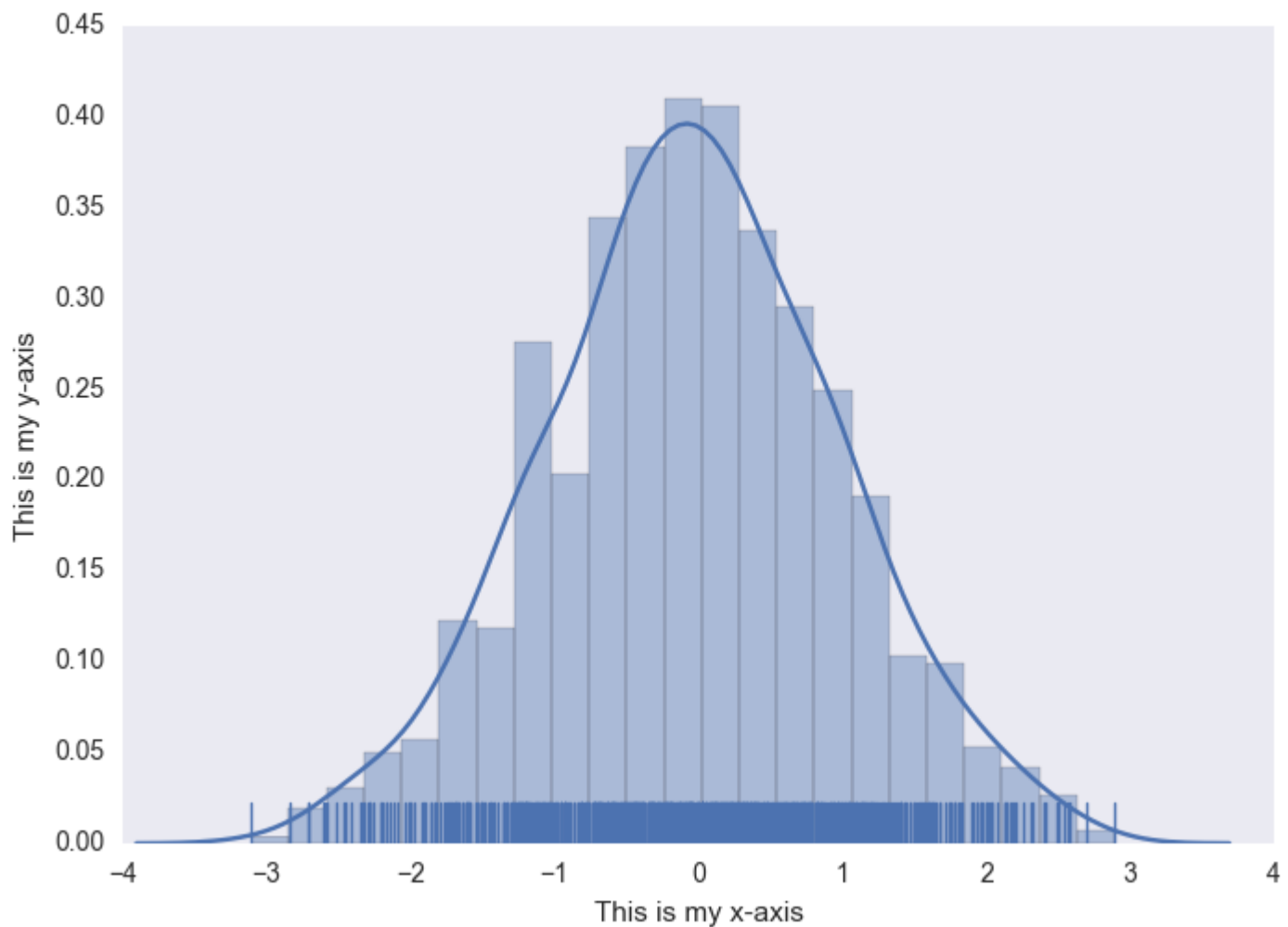



さらにボーナスとして、のmatplotlibコマンドをSeabornプロットにすることができます。ここでは、したヒストグラムにのタイトルをするをします。

```
# 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は、データの3Dビジュアライゼーションツールです。これは、ビジュアライゼーションツールキットまたは**VTK**をフードのでします。**MayaVI**は、**VTK**のパワーをして、さまざまな3Dプロットおよびフィギュアをすることができます。これは、のソフトウェアアプリケーションとして、またライブラリとしてもできます。**Matplotlib**とに、このライブラリは、**VTK**についてなくてもプロットをするためのオブジェクトプログラミングインターフェイスをします。

MayaVIは**Python 2.7x**シリーズでのみですすぐに**Python 3-x**シリーズでできることがされています**Python 3**でをするといくつかのがめられています

ドキュメントは[ここにあります](#)。いくつかのギャラリーのが[ここにあります](#)

ドキュメントから**MayaVI**をしてされたサンプルプロットをします。

```
# 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

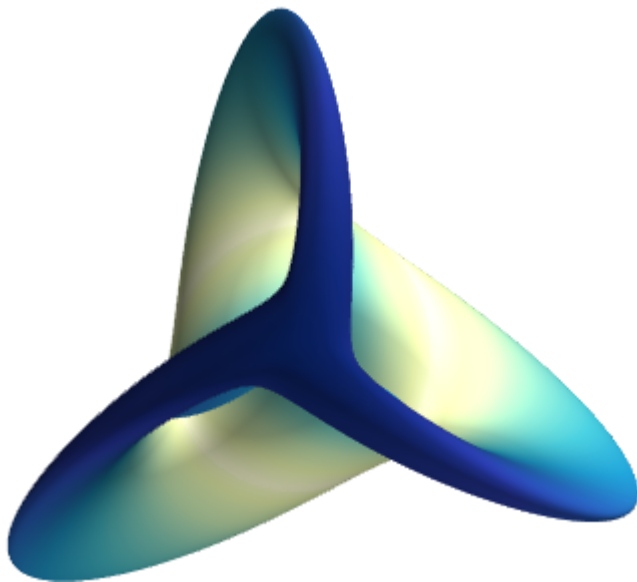
mlab.figure(fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))
u, v = mgrid[- 0.035:pi:0.01, - 0.035:pi:0.01]

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()

```



プロット

Plotlyはプロットとデータののためのプラットフォームです。 **Plotly**は、 **Python**、 **R**、 **JavaScript**、 **Julia**、 および**MATLAB**のライブラリとしてできます。また、これらのをしたWebアプリケーションとしてもできます。

ユーザは、ユーザに、プロットライブラリをインストールしてオフラインですることができます。このライブラリのインストールとオフラインが[ここにされています](#)。また、プロットはジュピターノートブックでもできます。

このライブラリをするには、ユーザーとパスワードのアカウントがです。これにより、クラウドのプロットやデータをするがえられます。

ライブラリには、がわずかにされており、1あたり250のプロットをできるようにされています。には、すべての、のプロットのダウンロード、さらにプライベートなデータストレージがあり

ます。については、メインページを[ご覧ください](#)。

ドキュメントとサンプルについては、[こちらをご覧ください](#)

ドキュメンテーションのサンプルのプロット

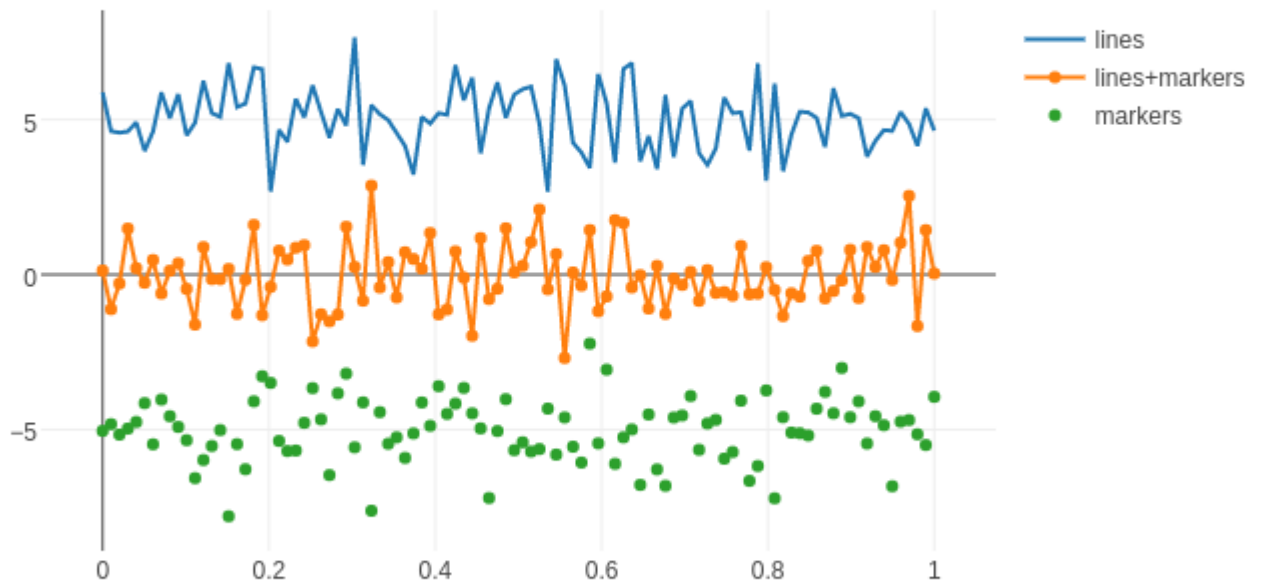
```
import plotly.graph_objs as go
import plotly as ply

# 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')
```



オンラインでPythonによるデータをむ <https://riptutorial.com/ja/python/topic/2388/pythonによるデータ>

50: Python ネットワーク

になPythonクライアントソケットの

Examples

もなPython ソケットクライアント/サーバの

サーバ

```
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
```

クライアント

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')
```

にSocketServer.pyをし、サーバーがsthをしるがっていることをしてから、クライアントがをサーバーにします。サーバーはsthをした、します。

シンプルなHTTPサーバの

ローカルネットワークでファイルをしたり、シンプルなWebサイトhttpとjavascriptをホストするために、PythonのみみSimpleHTTPServerモジュールをすることができます。PythonはあなたのPathになければなりません。ファイルがあるフォルダにし、のようにします。

python 2

```
$ python -m SimpleHTTPServer <portnumber>
```

python 3

```
$ python3 -m http.server <portnumber>
```

ポートがされていない、デフォルトポートは8000です。はのようになります。

0.0.0.0のポート8000でHTTPサービング

`http://hostipaddress:8000/`して、ローカルネットワークにされているのデバイスからファイルにアクセスできます。

`hostipaddress`はおそらく `192.168.xx`まるローカルIPアドレスです。

モジュールをするには、に `ctrl+c`.し `ctrl+c`.

TCPサーバーの

`socketserver` ライブラリーをしてTCPサーバーをすることができます。なエコーサーバがあります。

サーバ

```
from socketserver 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()
```

クライアント

```
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` サーバーをすると、なTCPサーバーをにできます。ただし、デフォルトでは、サーバはシングルスレッドであり、に1つのクライアントにしかできないことにしてください。のクライアントをするは、わりに `ThreadingTCPServer` をインスタンスするか、

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

UDPサーバーの

UDPサーバーは、`socketserver` ライブラリーをしてにできます。

なタイムサーバ

```
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()
```

テスト

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> sock = socket(AF_INET, SOCK_DGRAM)
>>> sock.sendto(b'', ('localhost', 5000))
0
>>> sock.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 5000))
```

な**HTTPServer**をスレッドでし、ブラウザをきます

プログラムがでWebページをしているにです。

```
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))
```


オンラインでPythonネットワークをむ <https://riptutorial.com/ja/python/topic/1309/pythonネットワーク>

51: Pythonのアンチパターン

Examples

なexcept

はですが、1つのexceptですべてを1にまとめることができます。

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

このでは、パターンの3つのをしています。

1. except5をいて、`KeyboardInterrupt`をむなをキャッチします。これにより、プログラムがしないことがあります。
2. exceptブロックはエラーをしません。つまり、`get_result`でがしたのか、`res`がリストであるのかをできません。
3. の、がであることをすると、かがひどくしました。`get_result`がすると、`res`はにされずに、exceptブロックの`res`へのは`NameError`をさせ、のエラーをにマスクします。

あなたがしようとしているのタイプについてにえてください。 [ページにみり](#)をえ、ながどのようなものかをる。

ののバージョンです

```
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)
```

たちはよりなをし、なをします。いくつかのが、にです。

あなたがプロセッサーののでするに

プログラムは、プロセッサのをびすことによって、をにできます。

ならば、それはをします。たとえば、のようになりまするvalue 1、そののすることはできません
None。

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

そして、のようになすることができます

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

print(x)
```

これはうまくいくが、intensive_fをびすというがあり、コードのが2になる。よりよいは、あら
かじめのりをすることです。

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

しかし、よりでもっとなやりは、をうことです。えば

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

ここではありません。assertをし、わりにAssertionErrorをassertがましいことがよくあります
。

キ—

これがつかるのは、キーにアクセスすることです。えば、

```
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)
```

with

```
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)
```

のでは、を2なければなりません。いであるため、そうするのにがにかかるかもしれません。2のは、を1だけするがあるため、くのプロセッサをできます。

これにわるは `dict.get(key, default)` をうことですが、くのでは、キーがしないにはもっとながになることがあります

オンラインでPythonのアンチパターンをむ <https://riptutorial.com/ja/python/topic/4700/pythonのアンチパターン>

52: Pythonのソースコードとバイトコードへのアクセス

Examples

のバイトコードをする

PythonインタプリタはPythonのマシンでコードをするにコードをバイトコードにコンパイルしますPythonの**バイトコード**とは

Pythonのバイトコードをするはのとおります

```
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)
```

disモジュールの**dis.dis**は、されたのコンパイルされたバイトコードをします。

のコードオブジェクトの

CPythonでは、オブジェクトのコードオブジェクトにアクセスできます。

`__code__`オブジェクトには、ののバイトコード `co_code` と、やなどのそののがまれています。

```
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__)
```

オブジェクトのソースコードをする

みみではないオブジェクト

Pythonオブジェクトのソースコードをするには**inspect**います。これはみみオブジェクトやにされたオブジェクトにしてはしません。これらについては、でするのがです。

random

モジュールからメソッド `random.randint` ソースコードをするは `random.randint` です。

```
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)
```

ドキュメントをするだけです

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

`random.randint` メソッドがされているファイルのフルパスをします。

```
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
```

インタラクティブにされたオブジェクト

オブジェクトがインタラクティブにされている、`inspect` はソースコードをできませんが、わりに `dill.source.getsource` をできます

```
# 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
```

みみオブジェクト

Pythonのみのソースコードは **c** でかかれており、Pythonのソースコード [Mercurial](https://www.python.org/downloads/source/) でホストされているか、<https://www.python.org/downloads/source/> からダウンロードをすることによってのみアクセスできます。

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

オンラインでPythonのソースコードとバイトコードへのアクセスをむ

<https://riptutorial.com/ja/python/topic/4351/pythonのソースコードとバイトコードへのアクセス>

53: Pythonのテンプレート

Examples

テンプレートをしたなデータプログラム

```
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))
```

```
Simon bought 2 candy for 8 dollar
```

テンプレートはベースののわりに\$ベースのをサポートしています。 **Substitute mapping**、**keywords**はテンプレートをし、しいをします。

マッピングとは、テンプレートのプレースホルダとするキーをつなオブジェクトのことです。ここでは、**price**と**qty**はプレースホルダです。キーワードはプレースホルダとしてもできます。のキーワードがするは、キーワードのプレースホルダがされます。

りの

"\$"デリミタはのものにできます。の

```
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))
```

あなたは[ここで](#)ドキュメントをむことができます

オンラインでPythonのテンプレートをむ <https://riptutorial.com/ja/python/topic/6029/pythonのテンプレート>

54: Pythonの

Pythonは、`threading`と`multiprocessing`のAPIがしているため、プログラマにとって2つののりえがになりました。

Examples

スレッドモジュール

```
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()
```

Iは**L**OCKをnterpreter GIL、または**G**のローブとしてられるものをするので、このようなCPythonのようなパイソンののでは、のは、スレッドをしてされません。

Pythonののれたをにします。

David BeazleyによるPythonのYouTube

マルチプロセッシングモジュール

```
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()
```

ここでは、新しいプロセスでされます。 Python VMの新しいインスタンスがコードをしているので、 GILはせず、のコアでがされます。

`Process.start` メソッドは、この新しいプロセスをし、 `args` をして `target` でされたをします。

`Process.join` メソッドは、プロセス `p1` と `p2` のをちます。

新しいプロセスは、パイソンのバージョンとコードがえはされている `platform` にじてなるされます

- Windowsは新しいプロセスをするために `spawn` をします。
- UNIXシステムと3.3よりのバージョンでは、 `fork` をしてプロセスがされます。
このメソッドは `fork` のPOSIXをしないため、にのマルチプロセッシングライブラリとするときにしないにつながることにしてください。
- UNIXシステムとバージョン3.4では、プログラムのに `multiprocessing.set_start_method` をして `fork`、 `forkserver` または `spawn` いずれかをって新しいプロセスをすることができます。
`forkserver` と `spawn` メソッドは `fork` よりもくなりますが、しないをけます。

POSIX forkの

マルチスレッドプログラムの `fork`、 `execve` をびすまでに `async-signal-safe` だけをにびすことができます。

`fork` をすると、のすべての `mutex` にしてくじでしいプロセスがされますが、 `MainThread` だけが `MainThread` されます。それがレースのなどにつながるがあるので、これはではありません。

- `MainThread` で `Lock` をし、それをあるでロックしようとしているのスレッドにすとしします。
`fork` にすると、しいプロセスはロックされたロックからします。ロックされたロックは、この新しいプロセスに2のスレッドがしないためされません。

のところ、こののはなPythonでは `multiprocessing` がにするためにはこらないはずですが、のライブラリとやりとりしている、こののがこり、システムがクラッシュすることがあります。

マルチプロセッシングプロセスでデータをす

データは2つのスレッドでされるときになのでをきこしてみるときみがするがあるとえます、スレッドでデータのやりとりをにするためにのオブジェクトがされました。のアトミックはすべてスレッドでできますが、に `Queue` にすることはにです。

```
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
```

のは、キューをするときは、のデータをすのではなく、 `tryexcept` ブロックにキューのデータをくことをします。ただし、スキャンサイクルをスキップするかどうかはありませんデータが `queue.Empty==True`、 `queue.Empty==False` をしているにキューにくことができます `iftry` ブロックとは

れるものできみアクセスをうことができます。なぜなら、`if`ステートメントはをキャッチするよりもにれているからです。

```
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 expensive.
    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
```

オンラインでPythonのをむ <https://riptutorial.com/ja/python/topic/3357/pythonの>

55: Pythonの

- `pickle.dumpobj`、`file`、`protocol = None`、*、`fix_imports = True`
- `pickle.loadfile`、*、`fix_imports = True`、`encoding = "ASCII"`、`errors = "strict"`

パラメーター

パラメータ	
オブジェクト	オープンファイルオブジェクトファイルへのobjのピクルされた
プロトコル	されたプロトコル、 0 -ASCII、 1 - いバイナリをするようにpicklerにします
ファイル	fileには、 <i>dump</i> メソッドとreadメソッドのロードのwriteメソッド _{wb} がです _{rb}

Examples

Pythonの

、リスト、、ネストされた、クラスインスタンスオブジェクトなどのオブジェクトは、コンピュータのメモリにされ、スクリプトがするとすぐにわれます。

`pickle`はデータを々のファイルににします。

オブジェクトのピクルされたはにすべてのにおいてバイトオブジェクトであるため、データをするために_{wb}ファイルを開き、`pickle`からデータをロードするために_{rb}をオープンするがあります。

データは、いかなるのものであってもよい。例えば、

```
data={'a':'some_value',
      'b':[9,4,7],
      'c':['some_str','another_str','spam','ham'],
      'd':{'key':'nested_dictionary'},
}
```

ストアデータ

```
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
```

データをみむ

```
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']}
```

のタイプはすることができます

1. なし、
2. 、
3. 、 バイト、バイト
4. pickableオブジェクトのみをむたフル、リスト、セット、および
5. モジュールのトップレベルでされたlambdaではなくdefを
6. モジュールのレベルでされたみみ
7. モジュールのトップレベルでされているクラス
8. **dict**または**getstate** をびしたのそのようなクラスのインスタンス

とみみのためのユーティリティ

ファイルとのでデータをする

```
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']
```

オンラインでPythonのをむ <https://riptutorial.com/ja/python/topic/7810/python>の

56: Pythonパッケージの

[pypa サンプルプロジェクト](#)には、`setup-tools`がしなければならないをすかつになテンプレート `setup.py`がまれています。

Examples

き

すべてのパッケージには、パッケージをする `setup.py` ファイルがです。

なパッケージのディレクトリをえてみましょう。

```
+-- package_name
|       |
|       +-- __init__.py
|
+-- setup.py
```

`__init__.py`には、`def foo(): return 100`というだけがまれています `def foo(): return 100`。

の `setup.py`はパッケージをします

```
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](#)はのPythonをすることなくパッケージのインストールをテストするのにです

```
$ 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
```

PyPIへのアップロード

setup.pyがにしたら [はじめにを](#)、パッケージをPyPIにアップロードできます。

.pypircファイルの

このファイルには、アカウントをするためのログインとパスワードがされています。、ホームディレクトリにされます。

```
# .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
```

パッケージをアップロードするにはtwineをするが**です**ので、インストールされていることをしてください。

```
$ pip install twine
```

testpypiへのとアップロードオプション

PyPIではアップロードされたパッケージをきすることはできませんので、まずtestpypiなどのテストサーバでデプロイメントをテストすることがです。このオプションについてします。 [カレンダーのバージョンング](#)や[セマンティックバージョンング](#)など、アップロードするにパッケージのバージョンングスキームをしてください。

testpypiでログインするか、しいアカウントをしてください。1のはではありませんが、めてののみがです。

```
$ python setup.py register -r pypitest
```

あなたのパッケージのルートディレクトリにある

```
$ twine upload dist/* -r pypitest
```

あなたのパッケージはあなたのアカウントからアクセスできるようになりました。

テスト

テストをする。testpypiまたはPyPIのいずれかからパッケージを`pip install`ください。

```
# 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
```

した、あなたのパッケージはなくともインポートです。PyPIへのアップロードのに、あなたのAPIをテストすることをするかもしれません。テストにパッケージにしたは、しないでください。あなたはそれをし、testpypiにアップロードして、テストすることができます。

PyPIへのとアップロード

twineがりがけられていることをしtwine。

```
$ pip install twine
```

ログインするか、[PyPI](#)でしいアカウントをしてください。

```
$ python setup.py register -r pypi
$ twine upload dist/*
```

それでおしまいあなたのパッケージは[されています](#)。

バグをしたは、しいバージョンのパッケージをアップロードするだけです。

ドキュメンテーション

あなたのパッケージになくともらかのドキュメントをめることをれないでください。PyPiはデフォルトのフォーマット `reStructuredText` をります。

Readme

パッケージにきなドキュメントがないは、のユーザーが `README.rst` ファイルにつをめて `README.rst`。ファイルがができたは、PyPiにそれをすようにするためにのファイルがです。

`setup.cfg` ファイルをし、の2をします。

```
[metadata]
description-file = README.rst
```

`Markdown` ファイルをパッケージにれようとすると、PyPiはなしでなテキストファイルとしてみます。

ライセンス

`OpenSource` `ライセンス` の1つをして `LICENSE.txt` ファイルをパッケージにれて、プロジェクトなどでパッケージをできるかどうか、またはライセンスでコードをできるかどうかをえることは、たいていのです。

よりみやすいで、いくつかのライセンスは `TL`; `DR` でされています。

パッケージをにする

あなたのパッケージがライブラリだけでなく、パッケージがインストールされているときにショーケースまたはスタンドアロンアプリケーションとしてできるコードがあるは、そのコードを `__main__.py` ファイルにれます。

`__main__.py` を `package_name` フォルダにれます。これでコンソールからすることができます

```
python -m package_name
```

な `__main__.py` ファイルがない、パッケージはこのコマンドでされず、このエラーがされます。

```
python package_name.__main__; というのモジュールはありません package_name.__main__;
'package_name'はパッケージであり、することはできません。
```

オンラインで `Python` パッケージのをむ <https://riptutorial.com/ja/python/topic/1381/pythonパッケージの>

57: Python リクエスト Post

き

HTTP POSTメソッドとそれにするRequestsのコンテキストでのPython Requestsモジュールのドキュメント

Examples

シンプルポスト

```
from requests import post

foo = post('http://httpbin.org/post', data = {'key':'value'})
```

なHTTP POSTをします。されたデータはほとんどののになりますが、キーののペアがもです。

ヘッダー

ヘッダーをすることができます。

```
print(foo.headers)
```

```
{'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'}
```

にヘッダーをすることもできます。

```
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'})
```

エンコーディング

エンコーディングは、ほほじでおよびできます。

```
print(foo.encoding)

'utf-8'

foo.encoding = 'ISO-8859-1'
```

SSL

は、デフォルトでドメインのSSLをします。これはオーバーライドすることができます

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, verify=False)
```

リダイレクション

のリダイレクトにいますhttpからhttpsなど。これもできます

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, allow_redirects=False)
```

ポストがリダイレクトされた、このにアクセスできます。

```
print(foo.url)
```

リダイレクトのなをすることができます

```
print(foo.history)
```

フォームエンコードデータ

```
from requests import post

payload = {'key1' : 'value1',
           'key2' : 'value2'
           }

foo = post('http://httpbin.org/post', data=payload)
```

ポストでフォームエンコードされたデータをすには、データをとてし、データパラメータとしてするがあります。

データがエンコードされていないようにするには、またはをdataパラメータにします。

データをにフォーマットするリクエストのjsonパラメータにをします

```
from requests import post

payload = {'key1' : 'value1', 'key2' : 'value2'}

foo = post('http://httpbin.org/post', json=payload)
```

ファイルアップロード

Requestsモジュールでは、`.read()` りされたコンテンツとはにファイルハンドルをするだけでよい。

```
from requests import post

files = {'file' : open('data.txt', 'rb')}

foo = post('http://http.org/post', files=files)
```

Filename、content_type、およびヘッダーもできます

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel',
{'Expires': '0'})}

foo = requests.post('http://httpbin.org/post', files=files)
```

は、`files` パラメータとしてされているり、ファイルとしてすることも`files` ます。

のファイル

1つのファイルとじょうにのファイルをすることができます。

```
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)
```

レスポンスコードは、ポストからできます。

```
from requests import post

foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.status_code)
```

されたデータ

されるデータへのアクセス

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.text)
```

の

になる`urllib3`の`response.HTTPResponse`オブジェクトにアクセスするがあるは、のようにしてうことができます。

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
```

```
res = foo.raw

print(res.read())
```

シンプルなHTTP

シンプルなHTTPは、のでできます。

```
from requests import post

foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

これはにはのではありません。

```
from requests import post
from requests.auth import HTTPBasicAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

HTTPダイジェスト

HTTPダイジェストはによくたでられますが、Requestsはこれにしてのオブジェクトをします。

```
from requests import post
from requests.auth import HTTPDigestAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0', 'natas0'))
```

カスタム

みみのメカニズムがでないもありますが、このを試してみてください。

がしいユーザーエージェントとのヘッダーをち、HTTPによってしいをする、サーバーはをけるようにされています。これをするには、カスタムクラスをし、RequestsのベースとなるAuthBaseをサブクラスするがあります。

```
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
```

これは、のコードでできます。

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

プロキシ

POSTは、ネットワークプロキシをするようにできます

HTTP / S プロキシ

```
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)
```

HTTPはのようになれます。

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

SOCKS プロキシ

ソックスプロキシをするには、インストールされたソックスプロキシがHTTPBasicAuthとによくたでされると、サードパーティの`requests[socks]`がです。

```
proxies = {
    'http': 'socks5://user:pass@host:port',
    'https': 'socks5://user:pass@host:port'
}

foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

オンラインでPythonリクエストPostをむ <https://riptutorial.com/ja/python/topic/10021/python> リクエストpost

58: PythonをCからびす

き

このドキュメントでは、CとPythonスクリプトのプロセスのをします。

のでは、NuGetマネージャをしてインストールできる**MongoDB.Bson**ライブラリをしてデータがシリアル化されています。

それのは、のJSONライブラリをできます。

に、プロセスのをします。

- はJSONにシリアルされ、なテキストファイルにされます

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

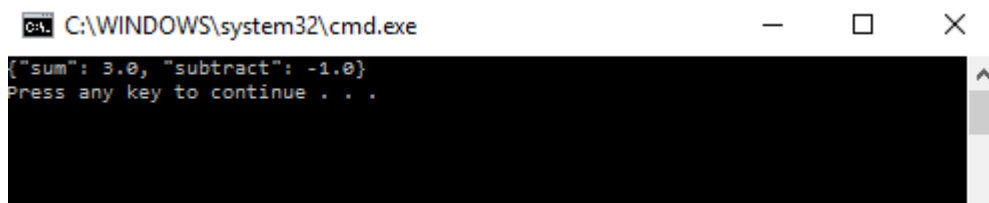
- Pythonインタプリタpython.exeは、テキストファイルからJSONをみみ、をりすPythonスクリプトをします。

```
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' ] ]
```

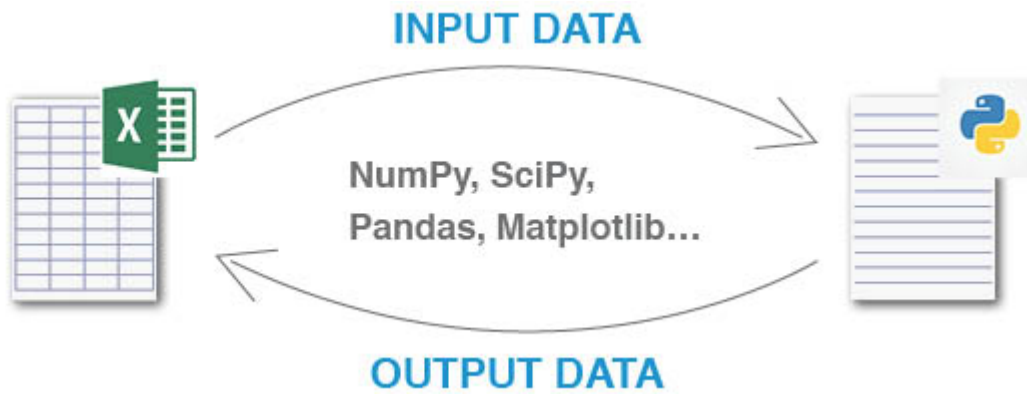
- Pythonスクリプトがされ、がJSONにシリアルされ、コマンドウィンドウにされます。

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



- CアプリケーションのJSONをみむ

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



は、ExcelスプレッドシートからPythonスクリプトをびすことができるのプロジェクトの1つで、CとPythonスクリプトのプロセスをしています。

このプロジェクトではExcelバインディングをC - Excelバインディングにしています。

ソースコードはGitHub [リポジトリ](#)にされています。

は、プロジェクトのをし、4つのなをするのにつWikiページへのリンクです。

-
- [の](#)
-
- [オブジェクトウィザード](#)
-

はあなたがそのとプロジェクトがにつことをっています。

Examples

Cアプリケーションによってびされる**Python**スクリプト

```
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 } )
```

Pythonスクリプトをびす**C**コード

```
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);
        }
    }
}

```

```
}
```

オンラインでPythonをCからびすをむ [https://riptutorial.com/ja/python/topic/10759/pythonをc-から
びす](https://riptutorial.com/ja/python/topic/10759/pythonをc-からびす)

59: PythonをしたWindowsサービスの

き

WindowsではヘッドレスプロセスUIなしを「サービス」とびます。コマンドコンソール、Powershell、タスクマネージャーの[サービス]タブなどのWindowsコントロールをして、などできます。いとして、Webアプリケーションなどのネットワークサービスをするアプリケーションや、々なバックグラウンドアーカイブタスクをするバックアップアプリケーションがあります。WindowsでPythonアプリケーションをしてサービスとしてインストールするはいくつかあります。

Examples

サービスとしてできるPython スクリプト

このでされるモジュールはpywin32 Python for Windows extensionsのです。 Pythonのインストールによっては、これをにインストールするがあります。

```
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)
```

これはなるです。おそらくのスクリプトをびすアプリケーションコードは、mainにります。

また、これをサービスとしてインストールするがあります。でのこのなは、 [サービスマネージャ](#) をすることです。これにより、サービスをインストールすることができ、サービスがするコマンドラインをするためのGUIがされます。Pythonの、これをうことができます。これは、にサービスをしします

```
nssm install MyServiceName c:\python27\python.exe c:\temp\myscript.py
```

my_script.pyはのスクリプトですが、mainでアプリケーションスクリプトやコードをびすようにしました。このサービスはPythonスクリプトをせず、Pythonインタプリタをし、コマンドラインのメインスクリプトにします。

また、Windows Serverリソースキットでされているツールをオペレーティングシステムのバージョンにしてサービスをするともできます。

Flask Webアプリケーションをサービスとしてする

これはなのバリエーションです。あなたは、アプリケーションのスクリプトをインポートして、サービスのmain()でrun()メソッドをびすだけでみます。この、WSGIRequestHandlerアクセスするのにより、マルチプロセッシングモジュールもしています。

```
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)
```

<http://stackoverflow.com/a/25130524/318488>からの

オンラインでPythonをしたWindowsサービスのをむ

<https://riptutorial.com/ja/python/topic/9065/python>をしたwindowsサービスの

60: Python - virtualenv

き

"virtualenv"は、されたPythonをするためのツールです。 Pythonのenvをにすることで、のプロジェクトでとされるを々のにします。「プロジェクトAはバージョン2.xxxにしますが、プロジェクトBは2.xxxがです」というジレンマをし、グローバルなサイトパッケージディレクトリをきれいでしやすくします。

"virtualenv"は、Pythonプロジェクトがとするパッケージをするのになすすべてのlibとbinをむフォルダをします。

Examples

インストール

pipen /apt-getでvirtualenvをインストールしてください

```
pip install virtualenv
```

または

```
apt-get install python-virtualenv
```

のがあるは、sudoをしてください。

```
$ cd test_proj
```

をする

```
$ virtualenv test_proj
```

のをするには、をアクティブにするがあります。

```
$ source test_project/bin/activate
```

あなたのvirtualenvをするには、に "deactivate"とタイプしてください

```
$ deactivate
```

Virtualenvにパッケージをインストールする

virtualenvのbinディレクトリをと、easy_installはvirtualenvのsite-packagesディレクトリにegg

とパッケージをれるようにされています。あなたのにアプリケーションをインストールするには

```
$ source test_project/bin/activate
$ pip install flask
```

では、ファイルはすべてローカルのvirtualenv site-packagesディレクトリにインストールされるため、sudoをするはありません。これはあなたのユーザーアカウントとしてされました。

そののな**virtualenv** コマンド

lsvirtualenv すべてののをします。

cdvirtualenv アクティブされているのディレクトリにします。たとえば、サイトパッケージをで
きます。

cdsitepackages とですが、site-packagesディレクトリにります。

Issitepackages site-packagesディレクトリのをします。

オンラインでPython - virtualenvをむ <https://riptutorial.com/ja/python/topic/9782/python---virtualenv>

61: setup.py

パラメーター

パラメータ	
name	あなたの。
version	あなたのバージョン。
packages	includeするPythonパッケージのリストモジュールをむディレクトリ。これはできますが、は <code>setuptools.find_packages()</code> びしがわりにされます。
py_modules	includeするのPythonモジュールのリストつまり、の <code>.py</code> ファイル。

Pythonパッケージングのについては、をしてください。

き

パッケージをくには、[パッケージングのユーザーガイド](#)があります。

Examples

setup.pyの

セットアップスクリプトは、Distutilsをしてモジュールを、およびインストールするすべてののです。ソフトウェアをしインストールするのがです。

foo.pyというファイルにまれているfooというモジュールをするだけなら、セットアップスクリプトはのようになにすることができます

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

このモジュールのソースディストリビューションをするには、のコードをむsetup.pyというsetupスクリプトをし、ターミナルからこのコマンドをします。

```
python setup.py sdist
```


sdistは、セットアップスクリプト`setup.py`とあなたのモジュール`foo.py`を含むアーカイブファイル。Unixではtarball、WindowsではZIPファイルを作ります。アーカイブファイルは`foo-1.0.tar.gz`または`.zip`とされ、`foo-1.0`ディレクトリにされます。

エンドユーザが`foo`モジュールをインストールしたいは、`foo-1.0.tar.gz`または`.zip`をダウンロードし、して`foo-1.0`ディレクトリからするだけです

```
python setup.py install
```

Pythonパッケージにコマンドラインスクリプトをする

Pythonパッケージのコマンドラインスクリプトはです。ユーザーがパッケージをインストールすると、そのパスでスクリプトをできるようにパッケージをできます。

コマンドラインスクリプト`hello_world.py`をつ`greetings`パッケージがあれば。

```
greetings/  
  greetings/  
    __init__.py  
    hello_world.py
```

のコマンドをすると、そのスクリプトをできます。

```
python greetings/greetings/hello_world.py
```

しかし、あなたがそうそれをしたい

```
hello_world.py
```

これをするには、のように`setup.py setup()`に`scripts`をします

```
from setuptools import setup  
setup(  
    name='greetings',  
    scripts=['hello_world.py']  
)
```

`greetings`パッケージをインストールすると、あなたのパスに`hello_world.py`がされます。

もう1つののは、エン트리ポイントをすることです。

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

このようにして、のようにするだけです。

```
greetings
```

setup.pyのソースメタデータの

`setuptools_scm`はGitまたはMercurialのメタデータを使ってパッケージのバージョンをべ、Pythonパッケージとそれにまれるパッケージデータをつけることができる、にされたパッケージです。

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

このではのをしています。バージョンのSCMメタデータのみをし、`find_packages()`びしをパッケージリストにきえるか、パッケージファインダのみをするには`use_scm_version=True`し
`use_scm_version=True`。

インストールオプションの

のでたように、このスクリプトのないはのとおりです。

```
python setup.py install
```

しかし、パッケージをインストールするなどのオプションがさらにあり、コードをしてインストールするなくテストすることができます。これはをしてわれます。

```
python setup.py develop
```

*Sphinx*ドキュメントのコンパイルや*Fortran*コードのなどのアクションをするは、のようなのオプションをできます。

```
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`と`finalize_options`は、そのがすように、`run`のでされます。

その、あなたはあなたのをびすことができます

```
python setup.py build_sphinx
```

オンラインで`setup.py`をむ <https://riptutorial.com/ja/python/topic/1444/setup-py>

62: Sqlite3モジュール

Examples

Sqlite3 - のサーバープロセスをとしません。

sqlite3モジュールは、GerhardHäringによってかれました。モジュールをするには、にデータベースをすConnectionオブジェクトをするがあります。ここでデータはexample.dbファイルにされます

```
import sqlite3
conn = sqlite3.connect('example.db')
```

memoryをすると、RAMにデータベースをできます。 Connectionをしたら、Cursorオブジェクトをし、そのexecuteメソッドをびしてSQLコマンドをできます。

```
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()
```

データベースからののとエラー

SQLite3データベースからをします。

クエリによってされたのをする

```
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
```

のするfetchoneメソッドをフェッチするには

```
print c.fetchone()
```

の、`fetchall`メソッドをします。

```
a=c.fetchall() #which is similar to list(cursor) method used previously
for row in a:
    print row
```

エラーは、`sqlite3.Error`をしようことができます

```
try:
    #SQL Code
except sqlite3.Error as e:
    print "An error occurred:", e.args[0]
```

オンラインでSqlite3モジュールをむ <https://riptutorial.com/ja/python/topic/7754/sqlite3モジュール>

63: sys

き

sysモジュールは、`sys.argv` コマンドラインパラメータや `sys.exit()` など、プログラムのにするとにアクセスして、プログラムフローののポイントからのプロセスをさせます。

モジュールにきれいにかけられているは、にはされているので、のではありません。

- **sys**モジュールをインポートし、のでできるようにする

```
import sys
```

- のを**sys**モジュールからののにインポートする

```
from sys import exit
```

すべての**sys**モジュールメンバーのについては、 [ドキュメント](#)をしてください。

Examples

コマンドラインの

```
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.
```

きくてされたプログラムでは、 [クリック](#) するなどのモジュールをしてコマンドラインをするのではなく、コマンドラインをすることにしてください。

スクリプト

```
# 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
```

エラーストリーム

```
# 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)
```

でプロセスをし、コードをす

```
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()
```

オンラインでsysをむ <https://riptutorial.com/ja/python/topic/9847/sys>

64: tempfile NamedTemporaryFile

パラメーター

パラメータ	
モード	ファイルを開くモード、デフォルト = w + b
	クローージャでファイルをするには、default = True
サフィックス	filename suffix、default = ''
プレフィックス	ファイル、デフォルト = 'tmp'
	tempfileをするdirname、デフォルト = なし
バッファサイズ	デフォルト = -1 オペレーティングシステムのデフォルトがされます

Examples

のなファイルをおよびそこにきむ

ファイルシステムにされるをつテンポラリファイルをすることができます。このファイルは、`name` プロパティでアクセスできます。このファイルは、UNIXシステムでは、クローズにするようにできます `delete` paramで、デフォルトはTrue。でオープンすることもできます。

は、きのファイルをしてき、「Hello World」ときます。そのファイルにします。テンポラリファイルのファイルパスには `name` でアクセスすることができます。このでは、`path` され、ユーザにされます。ファイルをじた、ファイルがかれ、ファイルのがみまれ、ユーザーにされます。

```
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()
```

```
/tmp/tmp6pireJ
Hello World!
```

オンラインでtempfile NamedTemporaryFileをむ

<https://riptutorial.com/ja/python/topic/3666/tempfile-namedtemporaryfile>

65: Unicode

Examples

エンコードとデコード

にUnicodeからバイトにエンコードします。ここでは、エンコーディングをします。

```
>>> u' '.encode('utf-8')  
'\xf0\x9f\x90\x8d'
```

もう1つのは、バイトからユニコードにデコードすることです。ここでは、エンコーディングかであるかをるがあります。

```
>>> b'\xf0\x9f\x90\x8d'.decode('utf-8')  
u'\U0001f40d'
```

オンラインでUnicodeをむ <https://riptutorial.com/ja/python/topic/5618/unicode>

66: Unicodeとバイト

- `str.encode`エンコーディング、`errors = "`
- `bytes.decode`エンコーディング、`errors = "`
- オープンファイル、モード、エンコーディング=なし

パラメーター

パラメータ	
エンコーディング	するエンコーディング ' <code>ascii</code> '、' <code>utf8</code> 'など
エラー	エラーモード、例えば、' <code>replace</code> 'はいをできえ、' <code>ignore</code> 'はいをするなどです。

Examples

Python 3では、`str`はユニコードので、`bytes`はのバイトのです。

```
type("f") == type(u"f")  # True, <class 'str'>
type(b"f")               # <class 'bytes'>
```

Python 2では、カジュアルなデフォルトで`raw`バイトのシーケンスであり、Unicodeは"`u`"というをつすべてのでした。

```
type("f") == type(b"f")  # True, <type 'str'>
type(u"f")               # <type 'unicode'>
```

Unicodeからバイトへ

Unicodeは`.encode(encoding)`バイトにできます。

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

py2では、デフォルトのコンソールエンコーディングは`sys.getdefaultencoding() == 'ascii'`で`utf-8`はないため、ののようにはできません。

```
>>> 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')
Tú13.55
```

エンコーディングがをできない、`UnicodeEncodeError`がします

```
>>> "£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)
```

ユニコードするバイト

バイトは、`.decode(encoding)` **Unicode**にできます。

バイトのシーケンスは、なエンコーディングをしてのみ**Unicode**にできます。

```
>>> b'\xc2\xa313.55'.decode('utf8')
'£13.55'
```

エンコードがをできない、`UnicodeDecodeError`がします。

```
>>> 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
```

エラーのエンコード/デコード

`.encode`と`.decode`にエラーモードがあります。

デフォルトは`'strict'`、エラーにがします。のモードはよりです。

エンコーディング

```
>>> "£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'
```

デコード

```
>>> 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'
```

ユニコードとバイトをうときは、エンコーディングをつぐにつことがであるということは、から
らかです。

ファイルI/O

バイナリモードでオープンされたファイル例えば、`'r'`または`'w'`は、をします。のエンコーディング
は`'utf8'`です。

```
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")
```

バイナリモード `'rb'`や`'wb'` でかれたファイルは、バイトをします。エンコーディングがないので
、エンコーディングはできません。

```
open(fn, mode='wb') # open file for writing bytes

# ERROR: cannot write string when bytes is expected:
open(fn, mode='wb').write("hi")
```

オンラインでUnicodeとバイトをむ <https://riptutorial.com/ja/python/topic/1216/unicodeとバイト>

67: virtualenvwrapperを使った

き

3つのプロジェクトA、プロジェクトB、プロジェクトC、プロジェクトA、プロジェクトBにはPython 3とライブラリがです。しかし、プロジェクトCでは、Python 2.7とライブラリがです。

そのためには、これらのプロジェクトをすることがベストプラクティスです。をするには、のを

Virtualenv、VirtualenvwrapperおよびConda

にはいくつかのオプションがありますが、virtualenvwrapperがもされています。

Examples

virtualenvwrapperを使ってをする

3つのプロジェクトA、プロジェクトB、プロジェクトC、プロジェクトA、プロジェクトBにはPython 3とライブラリがです。しかし、プロジェクトCでは、Python 2.7とライブラリがです。

そのためには、これらのプロジェクトをすることがベストプラクティスです。をするには、のを

Virtualenv、VirtualenvwrapperおよびConda

にはいくつかのオプションがありますが、virtualenvwrapperがもされています。

々はのためのいくつかのオプションがありますが、はいつも**virtualenvwrapper**をみます。

```
$ 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
```

これで、いくつかのソフトウェアをにインストールできます。

```
(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
```

lssitepackagesでいいパッケージをることができます

```
(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
```

にじてのをできます。

workonでをりえる

```
(python_3.6)$ workon python_3.5  
(python_3.5)$ echo $VIRTUAL_ENV  
/Users/salayhin/Envs/env1  
(python_3.5)$
```

virtualenvをするには

```
$ deactivate
```

オンラインで[virtualenvwrapper](https://riptutorial.com/ja/python/topic/9983/virtualenvwrapper)をつたをむ

<https://riptutorial.com/ja/python/topic/9983/virtualenvwrapper>をつた

68: Websockets

Examples

aiohttpを使ったシンプルなエコー

aiohttpは[aiohttp](#)します。

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())
```

aiohttpのラッパークラス

`aiohttp.ClientSession`は、カスタムWebSocketクラスとしてできます。

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()

```

オートバンを**Websocket**としてする

Autobahnパッケージは、Python Webソケットサーバーファクトリにできます。

[Python Autobahnパッケージのドキュメント](#)

インストールするには、はにterminalコマンドをします

Linuxの

```
sudo pip install autobahn
```

Windowsの

```
python -m pip install autobahn
```

に、なエコーサーバをPythonスクリプトですることができます

```
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
        otherwise 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:
        import asyncio
    except ImportError:
        '''Trollius = 0.3 was renamed'''
        import trollius as asyncio
    from autobahn.asyncio.websocket import WebSocketServerFactory
    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()

```

このでは、ポート9000のlocalhost127.0.0.1にサーバーがされています。これはリスンするIPとポートです。これはなです。これをすると、コンピュータにしているルーターはであれ、モデムからコンピュータのLANアドレスとポートフォワードをできます。に、googleをしてWAN IPをべると、WAN IP、ポート9000このではにWebSocketメッセージをするようにWebサイトをできます。

モデムのにポートすることはです。つまり、モデムにデイジーチェーンされているルータをしているは、モデムのをし、モデムからされたルータにします。されているモデムポート9000このではにがされていることをす。

オンラインでWebsocketsをむ <https://riptutorial.com/ja/python/topic/4751/websockets>

69: WebサーバーゲートウェイインターフェイスWSGI

パラメーター

パラメータ
start_response

start_response をするためにされる

Examples

サーバーオブジェクトメソッド

たちのサーバーオブジェクトには、アプリケーションパラメータがえられています。このパラメータは、びしなアプリケーションオブジェクトにすることができますのを。にヘッダーをきみ、にアプリケーションによってされたデータをシステムにきみます。

```
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()'
```

オンラインでWebサーバーゲートウェイインターフェイスWSGIをむ

<https://riptutorial.com/ja/python/topic/5315/webサーバーゲートウェイインターフェイス-wsgi->

70: Webブラウザモジュール

き

Pythonのなドキュメントによると、webbrowserモジュールはWebベースのドキュメントをユーザーにするためののインタフェースをします。このトピックでは、Webブラウザモジュールのなをしています。

- 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)

パラメーター

パラメータ	
webbrowser.open()	
URL	WebブラウザでくURL
しい	0はのタブのURLをき、1はしいウィンドウでき、2はしいタブで きます
オートライズ	Trueにすると、ウィンドウはのウィンドウのにされます
webbrowser.open_new()	
URL	WebブラウザでくURL
webbrowser.open_new_tab()	
URL	WebブラウザでくURL
webbrowser.get()	
をして	するブラウザ
webbrowser.register()	
URL	ブラウザ
コンストラクタ	なブラウザへのパス ヘルプ
インスタンス	webbrowser.get() メソッドからされたWebブラウザのインスタ ンス

のに、されたブラウザのをします。のは `webbrowser.get()` メソッドにすことができるで、のはブラウザタイプのクラスをリストします。

タイプ	クラス
'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

のブラウザでURLをく

URLをくには、 `webbrowser.open()` メソッドをします。

```
import webbrowser
webbrowser.open("http://stackoverflow.com")
```

ブラウザウィンドウがいては、したURLにしいタブがきます。ウィンドウがいていない、メソッドはオペレーティングシステムのデフォルトブラウザをき、パラメータのURLにします。openメソッドは、のパラメータをサポートしています。

- url - WebブラウザでくURL []
- new - 0はのタブでき、1はしいウィンドウをき、2はしいタブをきます [デフォルト0]
- autoraise - Trueにされている、ウィンドウはのアプリケーションのウィンドウのにされます
ブール [デフォルトはfalse]

のブラウザーのがこれらのコマンドをしているので、new autoraiseはほとんどしないことにしてください。

Webブラウザは、open_newメソッドをしてしいウィンドウでURLをこうとすることもできます。

```
import webbrowser
webbrowser.open_new("http://stackoverflow.com")
```

このメソッドはのブラウザではにされ、URLはしいタブでかれます。open_new_tabメソッドをして、しいタブをくことができます

```
import webbrowser
webbrowser.open_new_tab("http://stackoverflow.com")
```

なるブラウザでURLをく

webbrowserモジュールは、register() get() メソッドとget() メソッドをするなるブラウザもサポートしています。getメソッドは、のファイルのパスをしてブラウザコントローラをするためにされ、registerメソッドは、これらのファイルを、あらかじめのブラウザタイプがされているにするためにプリセットブラウザタイプにするためにされます。

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

ブラウザのをする

```
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/")
```

オンラインでWebブラウザモジュールをむ <https://riptutorial.com/ja/python/topic/8676/webブラウザ>

71: Windowsでvirtualenvwrapperを使ってをする

Examples

ウィンドウのvirtualenvwrapperによる

3つのプロジェクトA、プロジェクトB、プロジェクトC、プロジェクトA、プロジェクトBにはPython 3とライブラリがです。しかし、プロジェクトCでは、Python 2.7とライブラリがです。

そのためには、これらのプロジェクトをすることがベストプラクティスです。のpythonをするには、のうがあります。

ステップ1このコマンドでpython -m pip install -U pip python -m pip install -U pip

ステップ2に、コマンドをして "virtualenvwrapper-win"パッケージをインストールしますコマンドはWindows Power Shellをできます。

```
pip install virtualenvwrapper-win
```

3コマンドをしてしいをする mkvirtualenv python_3.5

ステップ4コマンドをしてをアクティブにする

```
workon < environment name>
```

virtualenvwrapperのなコマンド

```
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.
```

オンラインでWindowsでvirtualenvwrapperをつてをするをむ

<https://riptutorial.com/ja/python/topic/9984/windowsでvirtualenvwrapperをつてをする>

72: XMLの

XMLのすべてのが、されたツリーのとしてわるわけではありません。、このモジュールは、XMLコメント、およびのをスキップします。それにもかかわらず、XMLテキストからするのではなく、このモジュールのAPIをしてされたツリーはコメントとをつことができます。XMLをするときにインクルードされます。

Examples

ElementTreeをったオープニングとみみ

ElementTreeオブジェクトをインポートし、する.xmlファイルをき、ルートタグをします。

```
import xml.etree.ElementTree as ET
tree = ET.parse("yourXMLfile.xml")
root = tree.getroot()
```

ツリーをするはいくつかあります。はによるものです

```
for child in root:
    print(child.tag, child.attrib)
```

それのは、リストのようなのをすることができます

```
print(root[0][1].text)
```

のタグをでするには、.findまたは.findallします。

```
print(root.findall("myTag"))
print(root[0].find("myOtherTag"))
```

XML ファイルの

ツリーモジュールをインポートしてxmlファイルをき、xmlをする

```
import xml.etree.ElementTree as ET
tree = ET.parse('sample.xml')
root=tree.getroot()
element = root[0] #get first child of root element
```

オブジェクトは、フィールドの、のと、のによってできます

```
element.set('attribute_name', 'attribute_value') #set the attribute to xml element
element.text="string_text"
```

をするは、`Element.remove`メソッドをします。

```
root.remove(element)
```

`ElementTree.write`メソッドは、XMLオブジェクトをXMLファイルにするためにされます。

```
tree.write('output.xml')
```

XMLのと

ツリーのインポートモジュール

```
import xml.etree.ElementTree as ET
```

`Element`をしてXMLをする

```
p=ET.Element('parent')
```

をえるためにサブをするためにされる`SubElement`

```
c = ET.SubElement(p, 'child1')
```

`dump`は、xmlをダンプするためにされます。

```
ET.dump(p)
# Output will be like this
#<parent><child1 /></parent>
```

ファイルにするは、`ElementTree`をしてxmlツリーをし、ファイルにするには`write`メソッドをします

```
tree = ET.ElementTree(p)
tree.write("output.xml")
```

`Comment`は、xmlファイルにコメントをするためにされます。

```
comment = ET.Comment('user comment')
p.append(comment) #this comment will be appended to parent element
```

`iterparse` インクリメンタルをしてなXMLファイルをきみむ

によっては、なをるためにXMLファイルをロードしたくないもあります。これらのインスタスでは、するセクションを々にロードしてからするとすることができます。 `iterparse`をすると、XMLのにされているツリーをできます。

`ElementTree`オブジェクトをインポートします。

```
import xml.etree.ElementTree as ET
```

.xmlファイルをき、すべてのをします。

```
for event, elem in ET.iterparse("yourXMLfile.xml"):
    ... do something ...
```

また、タグやタグ、などのイベントのみをすることもできます。このオプションをしたのよう
に、「」イベントだけがされます。

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
    ... do something ...
```

ここでは、メモリのツリーからをクリアするをすなをします。

```
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 ...
```

XPathでXMLをする

バージョン2.7からElementTreeはXPathクエリをよりよくサポートしています。XPathは、データ
ベースをするためにされるSQLのようなxmlをナビゲートするためのです。findallとfindallの
find XPathをサポートしています。このでは、のxmlがされます

```
<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>
```

すべてのをする

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

title = 'The Magic of Color'のをする

```
tree.find("Books/Book[Title='The Colour of Magic']")  
# always use ' ' in the right side of the comparison
```

id = 5のをする

```
tree.find("Books/Book[@id='5']")  
# searches with xml attributes must have '@' before the name
```

2のをしてください

```
tree.find("Books/Book[2]")  
# indexes starts at 1, not 0
```

のをする

```
tree.find("Books/Book[last()]")  
# 'last' is the only xpath function allowed in ElementTree
```

すべてのをする

```
tree.findall("./Author")  
#searches with // must use a relative path
```

オンラインでXMLのをむ <https://riptutorial.com/ja/python/topic/479/xml>の

73: ZIPアーカイブをとってする

- zipfileをインポートする
- クラス zipfile。 **ZipFile** ファイル、モード='r'、 = ZIP_STORED、 allowZip64 = True

ZIPファイルではないファイルをとくと、 zipfile.BadZipFileがします。

Python 2.7では、これは zipfile.BadZipfile りが zipfile.BadZipfile られてい zipfile.BadZipfile。 このいは、Python 3.2でしいとともにっています

Examples

Zip ファイルをく

するには、 zipfile モジュールをインポートし、 ファイルをしします。

```
import zipfile
filename = 'zipfile.zip'
```

zipアーカイブをつたは、 **ファイルの**にていますが、 zip **ファイル**をいてオブジェクトをすると、 ファイルをじるにすることができます。

```
zip = zipfile.ZipFile(filename)
print(zip)
# <zipfile.ZipFile object at 0x0000000002E51A90>
zip.close()
```

Python 2.7とPython 3では3.2で、 with コンテキストマネージャをできます。 ファイルを "みり"モードでき、 ファイルのリストをしします

```
with zipfile.ZipFile(filename, 'r') as z:
    print(zip)
# <zipfile.ZipFile object at 0x0000000002E51A90>
```

Zipfileのをべる

zipファイルのをべるにはいくつかのがあります。 printdir をって、 さまざまなを stdout することができます

```
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.		

`namelist` メソッドでファイルのリストをすることもできます。ここでは、にリストをします

```
with zipfile.ZipFile(filename) as zip:
    print(zip.namelist())

# Out: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... etc. ...]
```

`namelist` ではなく、ファイルのタイムスタンプやファイルサイズなどをむ `ZipInfo` オブジェクトのリストをす `infolist` メソッドをびすことができます。

```
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
```

zip ファイルのをディレクトリにする

zip ファイルのすべてのファイルをする

```
import zipfile
with zipfile.ZipFile('zipfile.zip','r') as zfile:
    zfile.extractall('path')
```

のファイルをするには、メソッドをするには、パラメータとしてリストとパスをします

```
import zipfile
f=open('zipfile.zip','rb')
zfile=zipfile.ZipFile(f)
for cont in zfile.namelist():
    zfile.extract(cont,path)
```

しいアーカイブをする

きみモードでしいアーカイブオープンzip ファイルをします。

```
import zipfile
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

このアーカイブにファイルをするには、`write` メソッドをします。

```
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()
```

アーカイブにをきたいは、`writestr`メソッドをできます。

```
str_bytes="string buffer"  
new_arch.writestr('filename_string_in_archive.txt',str_bytes)  
new_arch.close()
```

オンラインでZIPアーカイブをとってするをむ <https://riptutorial.com/ja/python/topic/3728/zip> アーカイブをとってする

74: イテレータとイテレータ

Examples

イテレータとイテレータジェネレータ

iterableは、イテレータをすことができるオブジェクトです。 `__iter__` メソッドをち、イテレータをすのオブジェクトはすべてです。これは `__getitem__` メソッドをするのないオブジェクトでもあります。 - このメソッドはインデックスがゼロからまり、インデックスがもはやでないときに `IndexError` させることができます。

Pythonの `str` クラスは、 `__getitem__` iterable のです。

は、あるオブジェクトで `next(*object*)` をびすと、シーケンスののをすのオブジェクトです。さらに、 `__next__` メソッドをつオブジェクトはすべてイテレータです。イテレータはイテレータを `StopIteration` に `StopIteration` を `StopIteration` ため、このではできません。

なクラス

なクラスは `__iter__` と `__next__` メソッドをします。なクラスの

```
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())
```

`collections` モジュールからクラスをインスタンスして、これをよりよくしようとしています。

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__
```

のようにして、Python 2のなクラスにするPython 3のをします。

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
```

これらはともイテレータであり、ループすることができます

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

ジェネレータはイテレータをするなです。ジェネレータはイテレータであり、イテレータはです。

りしなもの

Iterableは、アイテムが1つずつにされるものであればどれでもかまいません。みみのPythonコレクションはりしです

```
[1, 2, 3]      # list, iterate over items
(1, 2, 3)      # tuple
{1, 2, 3}      # set
{1: 2, 3: 4}    # dict, iterate over keys
```

ジェネレータはiterablesをします

```
def foo(): # foo isn't iterable yet...
    yield 1

res = foo() # ...but res already is
```

iterableをする

```
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]

```

iterableの1つののみをする

してのをし、それがのものであることをしてください

```

a, = iterable

def foo():
    yield 1

a, = foo() # a = 1

nums = [1, 2, 3]
a, = nums # ValueError: too many values to unpack

```

を1つつする

イテレータをiterableするために`iter()`ビルトインからし、`next()`をして、`StopIteration`がをすまでを1つつします。

```

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

```

イテレータはリエントラントではありません

```

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()

```

オンラインでイテレータとイテレータをむ <https://riptutorial.com/ja/python/topic/2343/イテレータとイテレータ>

75: インタプリタコマンドラインコンソール

Examples

なヘルプの

`help` がなしでコンソールでびされると、Pythonはインタラクティブなヘルプコンソールをし、Pythonのモジュール、シンボル、キーワードなどについてることができます。

```
>>> 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".
```

の

コンソールののののをするには、アンダースコア_します。

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

このアンダースコアは、をもたらずPythonをしているにのみされます。またはforループをしてもはされません。がをさせる、_はされません。

```
>>> "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'
```

こののはインタラクティブなPythonインタプリタでのみであることを覚えておいてください。スクリプトをすることはできません。

Pythonコンソールをく

Pythonのプライマリバージョンのコンソールは、Windowsコンソールに`py`をするか、のプラットフォームの`python`をすることでくことができます。

```
$ 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.
>>>
```

のバージョンをおちの、デフォルトではファイルはそれぞれ`python2`または`python3`マッブされます。

これはもちろんPATHにあるPythonのファイルにします。

PYTHONSTARTUP

PythonのコンソールにPYTHONSTARTUPというをすることができます。 Pythonコンソールにると、このファイルがされ、にされているモジュールをにインポートするなどのなをコンソールにすることができます。

PYTHONSTARTUPがこれをむファイルのにされている、

```
print("Welcome!")
```

に、Pythonコンソールをくと、このながられます。

```
$ 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!
>>>
```

コマンドラインの

Pythonには、`py`にすことができるさまざまなコマンドラインスイッチがあります。 `py --help`すると、Python 3.4でこのがられます。

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.
```

オブジェクトにするヘルプの

Pythonコンソールには、やオブジェクトにするをするためにできるしい`help`されています。

の、`help`がそのシグネチャとそのdocstring `help`します。

```
>>> 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.
```

オブジェクトの、`help`はオブジェクトのdocstringとそのオブジェクトがつさまざまなメンバをリストします。

```
>>> 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...
```

オンラインでインタプリタコマンドラインコンソールをむ

<https://riptutorial.com/ja/python/topic/2473/インタプリタ-コマンドラインコンソール->

76: インデックスとスライス

- `obj` [ステップ]
- スライス
- スライス、`[`、ステップ]

パラメーター

パラメーター	
<code>obj</code>	「サブオブジェクト」をするオブジェクト
<code>start</code>	サブオブジェクトをする <code>obj</code> のインデックスPythonはゼロインデックスされていることにしてください。つまり、 <code>obj</code> のアイテムにはインデックスが <code>0</code> 。したのデフォルトは <code>0</code> です。
<code>stop</code>	サブオブジェクトをさせる <code>obj</code> のなインデックス。すると、デフォルトは <code>len(obj)</code> ます。
<code>step</code>	すべての <code>step</code> のみをできます。すると、デフォルトは <code>1</code> ます。

ユニコードが`1`のでされていることをして、をのとしてすることによって、をスライスするのをシーケンスをスライスするとすることができます。

では、`[start, end)`きのをするようスライスすることがえられます。つまり、がまれています、わりはありません。のオープンなは、そのする`len(x[:n]) = n`ここで、`len(x) >= n`、にじられているはしながら`x[n:n+1] = [x[n]]`ここで、`x`は`len(x) >= n`リストであるため、けとスライスのがたれます。

Examples

スライス

例えば、リストなどの、Pythonではデータのまたはリストをスライスしてすことができます。

スライシングのフォーマット

```
iterable_name[start:stop:step]
```

ここで、

- `start`はスライスのインデックスです。デフォルトは`0`のインデックス

- スライスのインデックスを1つ`stop`します。デフォルトは`len(iterable)`です。
- `step`はステップサイズですのでよりしくしています

```
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))
```

さらに、のいずれかをされたステップサイズですることができます。

```
a[::2]     # "ace" (every 2nd element)
a[1:4:2]   # "bd" (from index 1, to index 4 (excluded), every 2nd element)
```

インデックスはのにすることができます。この、インデックスはシーケンスのからされます

```
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()))
```

ステップサイズはのにすることもできます。この、スライスはリストをにりします。

```
a[3:1:-1]  # "dc" (from index 2 to None (default), in reverse order)
```

このは、な

```
a[::-1]    # "fedcba" (from last element (default len()-1), to first, in reverse order(-1))
```

のステップでは、デフォルトの`end_index`は`None` <http://stackoverflow.com/a/12521981> であること
にしてください。

```
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))
```

のいコピーをる

のコピーをするなは、のにのでをするのではなくです

```
arr[:]
```

をべてみましょう。 `[:]`は、 `start`、 `end`、 および `slice` がすべてされていることをし `slice`。 それらはデフォルトで `0`、 `len(arr)`、 `1` とそれぞれされています。つまり、しているには、からまで `arr` すべてのがあります。

には、これはのようになります。

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)      # ['a', 'b', 'c', 'd']
print(copy)     # ['a', 'b', 'c']
```

このとおり、 `arr.append('d')` は `arr` に `d` をしましたが、 `copy` はされませんでした。

これは、いコピーをし、 `arr.copy()` とじであることにしてください。

オブジェクトの

スライスをする、 `str`、 `list`、または `tuple` またはには `step` パラメータでスライスをするのコレクションオブジェクトをににすることができます。ののにもじことがてはありますが、をさせるをにします

```
s = 'reverse me!'
s[::-1]    # '!em esrever'
```

すぐにをてみましょう。 `[::-1]` は、スライスがのからまでであることをします `start` と `end` がされているため `-1` は、をにするがあります。

カスタムクラスのインデックス `__getitem__`、 `__setitem__`、 `__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]

```

これにより、アクセスのスライスとインデックスがになります。

```

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

```

のとでは、カンマリインデクシングスライスなしのみがです。

```

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]

```

スライスのりて

スライスをするのきれいなは、スライスのりてです。Pythonでは、しいスライスをりてて、リストのいスライスを1のできえることができます。

つまり、リストがあるは、1つのりてでのメンバーをきえることができます。

```

lst = [1, 2, 3]
lst[1:3] = [4, 5]
print(lst) # Out: [1, 4, 5]

```

りてがサイズにもしないようにするがあります。いスライスをサイズのなるしいスライスにきえるは、のようにします。

```

lst = [1, 2, 3, 4, 5]
lst[1:4] = [6]
print(lst) # Out: [1, 6, 5]

```

また、のスライシングをして、リストをきえるなどのをうこともできます。

```
lst = [1, 2, 3]
lst[:] = [4, 5, 6]
print(lst) # Out: [4, 5, 6]
```

または、の2だけ

```
lst = [1, 2, 3]
lst[-2:] = [4, 5, 6]
print(lst) # Out: [1, 4, 5, 6]
```

スライスオブジェクト

スライスはそれのオブジェクトであり、みみの`slice()`をしてにすることができます。スライスは、コードをみやすくしてをするためにできます。

```
>>> 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']
```

なインデックス

Pythonリストは0ベースです。つまり、リストののはインデックス₀アクセスできます

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

あなたは、インデックスによって、リストの2のにアクセスすることができます₁インデックスにより、の₂というように

```
print(arr[1])
>> 'b'
print(arr[2])
>> 'c'
```

また、のインデックスをして、リストのからにアクセスすることもできます。えは。インデックス₋₁はリストののをえ、インデックス₋₂はリストのから2のをえま

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

リストにしないインデックスにアクセスしようとすると、`IndexError`がします

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

オンラインでインデックスとスライスをもむ <https://riptutorial.com/ja/python/topic/289/インデックスとスライス>

77: インデント

Examples

インデントエラー

はをしてででなければならない。なげは、 `IndentationError` きこしたり、プログラムがせぬことを
するとなります。のでは、 `IndentationError` ます。

```
a = 7
if a > 5:
    print "foo"
else:
    print "bar"
print "done"
```

または、コロンにくがインデントされていない、 `IndentationError` もします。

```
if True:
print "true"
```

していないインデントをすると、 `IndentationError` がします。

```
if True:
    a = 6
        b = 5
```

げをれると、がわれるがあります。このでは、 `False` わりに `None` がされます。

```
def isEven(a):
    if a%2 ==0:
        return True
        #this next line should be even with the if
        return False
print isEven(7)
```

な

Pythonの、Guido van Rossumはげにするのグループについていました。このは、「[デザインとヒストリーPythonのFAQ](#)」ののセクションでされています。コロンは、`:`、するためにされるインデントコードブロックをし、ののように、。

```
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])

```

スペースやタブ

インデントは4スペースですが、しているり、タブまたはスペースをできます。Python 3でエラーがし、Python 2でエラーがするがあるため、**Python**ではタブとスペースをしないでください。

インデントがどのようにされるか

は、されるにツールによってされます。

レキシカルアナライザは、インデントレベルをするためにスタックをします。のスタックには0がまれています。これはのです。ネストされたブロックがまると、しいインデントレベルがスタックにプッシュされ、"INDENT"トークンがパーサーにされるトークンストリームにされます。1つののにの"INDENT"トークンがすることはありません `IndentationError`。

よりさなげレベルでがつかると、しいげレベルにしいがになるまでがスタックからポップされます。つからないはエラーがします。ポップされたにして、「DEDENT」トークンがされます。らかに、の「DEDENT」トークンがしてするがあります。

アナライザは、とコメントのみをむをスキップし、それらのために「インデント」または「デデント」トークンをしません。

ソースコードのには、スタックにされたインデントレベルごとに "DEDENT" トークンがされ、0だけがされます。

えば

```

if foo:
    if bar:
        x = 42
else:
    print foo

```

のようにされる

```
<if> <foo> <:>           [0]
<INDENT> <if> <bar> <:>      [0, 4]
<INDENT> <x> <=> <42>        [0, 4, 8]
<DEDENT> <DEDENT> <else> <:> [0]
<INDENT> <print> <foo>      [0, 2]
<DEDENT>
```

パーサーは、ブロックリとして "INDENT"と "DEDENT"トークンをします。

オンラインでインデントをむ <https://riptutorial.com/ja/python/topic/2597/インデント>

78: ウルリブ

Examples

HTTP GET

Python 2.x 2.7

Python 2

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

`urllib.urlopen()` をうとレスポンスオブジェクトがされます。これはファイルとにうことができます。

```
print response.code
# Prints: 200
```

`response.code` は、httpのりをします。 200はOK、404はNotFoundなど

```
print response.read()
'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack. etc'
```

`response.read()` と `response.readlines()` は、 リクエストからされたのhtmlファイルをみむためにできます。これらのメソッドは、 `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>'
```

モジュールはPython 3.xにされましたが、ユースケースはにじです。 `urllib.request.urlopen` はのファイルのようなオブジェクトをします。

HTTP POST

POSTデータには、エンコードされたクエリをデータとしてurlopenにします。

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'
```

コンテンツタイプのエンコーディングによってバイトをデコードする

けたバイトはしいエンコーディングでデコードしてテキストとしてするがあります

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)
```

オンラインでウルリブをむ <https://riptutorial.com/ja/python/topic/2645/ウルリブ>

79: オーディオ

Examples

Pygletでオーディオ

```
import pyglet
audio = pyglet.media.load("audio.wav")
audio.play()
```

は、「[pyglet](#)」をしてください。

WAVファイルの

winsound

- Windows

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```

-
- モノラル/ステレオをサポート
 - /をサポートしていません

```
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)
```

のサウンドファイルをpythonとffmpegでする

```
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()
```

- <http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is-there-even-a-difference>
- [ffmpeg](#)、[libav](#)、[avconv](#)のいとはですか

Windowsのビーブをらす

Windowsではなインターフェイスがされているため、`winsound`モジュールをして、したとでのビーブをすることができます。

```
import winsound
freq = 2500 # Set frequency To 2500 Hertz
dur = 1000 # Set duration To 1000 ms == 1 second
winsound.Beep(freq, dur)
```

オンラインでオーディオをむ <https://riptutorial.com/ja/python/topic/8189/オーディオ>

80: オーバーロード

Examples

マジック/ダンダーメソッド

PythonのMagiceddouble-underscoreのとしてのdunderともばれるメソッドは、ののオーバーロードとのをたします。それらは、クラスがまたは2のオペランドとしてされるときに、クラスののいをするをにします。また、みみによってびされるときもします。

この2ベクトルのをえてみましょう。

```
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)
```

さまざまで`Vector`クラスのインスタンスをにできるようになりました。

```
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

```

コンテナおよびシーケンス

キーまたはインデックスによるへのアクセスをサポートするコンテナタイプをエミュレートすることはです。

メモリをするためにゼロのだけをするスパースリストのこのなをえてみましょう。

```

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

```

に、のlistのようなsparselist listすることができます。

```

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

```

びしな

```

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

```

されていないの

あなたのクラスがするののためののオーバーロードされたをしていない、それはすべき `return NotImplemented` があることにして、**な**とじではない、`NotImplementedError`。これにより、Pythonはのメソッドをしてをうことができます

`NotImplemented` がされると、インタプリタは、にじて、のまたはのフォールバックにして、されたをします。されたのすべてが `NotImplemented` した、インタプリタはなをさせます。

たとえば、`x + y`と、`x.__add__(y)` がされていない、`y.__radd__(x)` がわりにされます。

```

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__

```

これがされたメソッドであるため、すべてのケースでされるをるために `__add__` と `__radd__` をする `__add__` あります。にも、このなではともじことをしているので、たちはなことをすることができます。

で

```
>>> 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
```

オペレータのオーバーロード

は、クラスでオーバーロードできるとなメソッド、でされているのです。

としての`other`のはではありませんが、となされます。

オペレーター		
+	<code>__add__(self, other)</code>	<code>a1 + a2</code>
-	<code>__sub__(self, other)</code>	<code>a1 - a2</code>
*	<code>__mul__(self, other)</code>	<code>a1 * a2</code>
@ の	<code>__matmul__(self, other)</code>	<code>a1 @ a2</code> <i>Python 3.5</i>
/	<code>__div__(self, other)</code>	<code>a1 / a2</code> <i>Python 2のみ</i>
/	<code>__truediv__(self, other)</code>	<code>a1 / a2</code> <i>Python 3</i>
// フロアディビジョン	<code>__floordiv__(self, other)</code>	<code>a1 // a2</code>
% モジュロ/リ	<code>__mod__(self, other)</code>	<code>a1 % a2</code>
** パワー	<code>__pow__(self, other[, modulo])</code>	<code>a1 ** a2</code>
<< ビットごとのシフト	<code>__lshift__(self, other)</code>	<code>a1 << a2</code>
>> ビットのシフト	<code>__rshift__(self, other)</code>	<code>a1 >> a2</code>
& ビットAND	<code>__and__(self, other)</code>	<code>a1 & a2</code>
^ ビットの	<code>__xor__(self, other)</code>	<code>a1 ^ a2</code>
ビットOR	<code>__or__(self, other)</code>	<code>a1 a2</code>

オペレーター		
-	<code>__neg__(self)</code>	<code>-a1</code>
+	<code>__pos__(self)</code>	<code>+a1</code>
~ビットのNOT	<code>__invert__(self)</code>	<code>~a1</code>
<よりさい	<code>__lt__(self, other)</code>	<code>a1 < a2</code>
<=よりさいかしい	<code>__le__(self, other)</code>	<code>a1 <= a2</code>
==しい	<code>__eq__(self, other)</code>	<code>a1 == a2</code>
!=しくない	<code>__ne__(self, other)</code>	<code>a1 != a2</code>
>よりきい	<code>__gt__(self, other)</code>	<code>a1 > a2</code>
>=よりきいまたはしい	<code>__ge__(self, other)</code>	<code>a1 >= a2</code>
[index]	<code>__getitem__(self, index)</code>	<code>a1[index]</code>
inには	<code>__contains__(self, other)</code>	<code>a2 in a1</code>
(*args, ...)びし	<code>__call__(self, *args, **kwargs)</code>	<code>a1(*args, **kwargs)</code>

`__pow__` `modulo` オプションのパラメータは、`pow` みみでのみされます。

バイナリにするメソッドには、する `__r` でまる `__r` などの「しい」メソッドがあり `__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
```

する `__i` バージョンとに、`__i` でまり `__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
```

これらのメソッドにはなものはないので、ののくの、ライブラリの、さらにはサードパーティのモジュールでさえ、オブジェクトをにキャストするメソッドやオブジェクトのプロパティをチェックするメソッドのようなメソッドをします。たとえば、みみの`str()`は、オブジェクトの`__str__`メソッドがするにそれをびします。これらののをにします。

intキャスト	<code>__int__(self)</code>	<code>int(a1)</code>
	<code>__abs__(self)</code>	<code>abs(a1)</code>
strキャストする	<code>__str__(self)</code>	<code>str(a1)</code>
unicodeへのキャスト	<code>__unicode__(self)</code>	<code>unicode(a1)</code> Python 2のみ
	<code>__repr__(self)</code>	<code>repr(a1)</code>
boolへのキャスト	<code>__nonzero__(self)</code>	<code>bool(a1)</code>
の	<code>__format__(self, formatstr)</code>	<code>"Hi {:abc}".format(a1)</code>
ハッシング	<code>__hash__(self)</code>	<code>hash(a1)</code>
さ	<code>__len__(self)</code>	<code>len(a1)</code>
	<code>__reversed__(self)</code>	<code>reversed(a1)</code>
	<code>__floor__(self)</code>	<code>math.floor(a1)</code>
	<code>__ceil__(self)</code>	<code>math.ceil(a1)</code>

コンテキストマネージャには`__enter__`や`__exit__`なメソッドもあります。

オンラインでオーバーロードをむ <https://riptutorial.com/ja/python/topic/2063/オーバーロード>

81: オペレータの

き

Pythonにはがされているため、あいまいなでにされるがまります。たとえば、 $3 * 2 + 7$ では、の3が2でされ、が7にされて13になります。 $*$ は $+$ よりもいをつため、のはされません。

は、によるのリストと、それらがうことのなです。

Pythonのドキュメントから

のは、もいのバインディングからもいほとんどのバインディングまで、Pythonののをまとめたものです。じボックスのはじをちます。がにえられていないり、はバイナリです。じボックスグループのは、からへのとがすべてじで、からにグループされたをむテストをめて、からへのです。

オペレーター	
ラムダ	ラムダ
if - else	
または	ブールOR
そして	ブールAND
xでない	ブールNOT
<, <=, >, >=, ==, !=	メンバーシップテストとアイデンティティテストをむ
	ビットOR
^	ビットXOR
そして、	ビットAND
<<, >>	シフト
+, -	しとき
*, /, //	, , [8]
+x, -x, ~x	, , ビットNOT
**	[9]

オペレーター	
x[index]、x[indexindex]、xarguments ...、 x.attribute	サブスクリプション、スライシング、コール 、
...、[...]、{キー...}、...	バインディングまたはタプル、リスト、

Examples

シンプルなの

PythonはPEMDASルールにいます。PEMDASは、、、けと、とのです。

```
>>> 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
```

エクストラなルールはしますが、**ずしもそうではありません**。

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

オンラインでオペレータのをむ <https://riptutorial.com/ja/python/topic/5040/オペレータの>

82: カウント

Examples

`iterablecollections.Counter`のすべてのアイテムのすべてのをえる

```
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
```

`collections.Counter`は、のにでき、すべてののすべてのをえます。

1つののは、`dict`またはの`collections.Mapping`ようなクラスがされている、それらはカウントされず、これらのをつカウンタがされます。

```
Counter({"e": 2})
# Out: Counter({'e': 2})

Counter({"e": "e"})      # warning Counter does not verify the values are int
# Out: Counter({'e': "e"})
```

もなをする-`sollections.Counter.most_common`

`collections.Counter`では`Mapping`のキーをえることはできませんが、をえることができます

```
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})
```

もなは、`most_common` -methodによって`most_common`です。

```
# 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)]
```

シーケンスの1つのアイテムのをカウントするlist.countおよび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
```

ののをえるstr.count

```
astring = 'thisisashorttext'
astring.count('t')
# Out: 4
```

これは、1よりいにしてもします。

```
astring.count('th')
# Out: 1
astring.count('is')
# Out: 2
astring.count('text')
# Out: 1
```

これはcollections.Counterではです。のだけをカウントするカウンタ

```
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})
```

numpyでをカウントする

ののをえる。これはします

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

は、ブールステートメントが、されたのすべてののが1であり、そのすべてがゼロであるをすることです。これらをする、がまります。これは、のまたはdtypeのにしてします。

numpyでのすべてののをカウントするためにがする2つのメソッドがあります。ユニークでです。Uniqueはをにしますが、bincountはののみをむ1dでのみします。

```
>>> 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]
```

データががのは、にnumpyメソッドをしてデータをメソッドにするがはるかにです。

オンラインでカウントをむ <https://riptutorial.com/ja/python/topic/476/カウント>

83: カスタムエラー/をさせる

き

Pythonにはみみのがいくつかあります。みみのは、プログラムのかがっているときにエラーをするようにします。

ただし、にったカスタムをするがじることがあります。

Pythonでは、ユーザーはしいクラスをすることによってそのようなをできます。このクラスは、Exceptionクラスからまたはにするがあります。みみのほとんどは、このクラスからもしています。

Examples

カスタム

ここでは、ExceptionクラスからしたCustomErrorというユーザーのをしました。このしいは、の
とに、raiseとオプションのエラーメッセージをしてびすことができます。

```
class CustomError(Exception):
    pass

x = 1

if x == 1:
    raise CustomError('This is custom error')
```

```
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
```

カスタムをキャッチ

このは、カスタムをキャッチするをしています。

```
class CustomError(Exception):
    pass

try:
    raise CustomError('Can you catch me ?')
except CustomError as e:
    print ('Caught CustomError :{}'.format(e))
except Exception as e:
    print ('Generic exception: {}'.format(e))
```

```
Caught CustomError :Can you catch me ?
```

オンラインでカスタムエラー/をさせるをむ <https://riptutorial.com/ja/python/topic/10882>/カスタムエラー-をさせる

84: ガベージコレクション

Pythonのガベージコレクタ3.5は、そのとなるなカウントです。オブジェクトたとえば `a = myobject` へのををするたびに、そのオブジェクト `a = myobject` のカウントがインクリメントされます。がされるたびに、カウントがし、カウントが0にすると、そのオブジェクトへのををするものはないことがかり、することができます。

Pythonのメモリがどのようにするかについてのなの1つは、`del` キーワードはオブジェクトのメモリをすることです。じゃない。には、`del` キーワードはにオブジェクト `refcount` をらすだけです。つまり、`refcount` が0にするのになをびすと、オブジェクトでガベージコレクションがわれるがあります。

Pythonはオブジェクトをめてとするときにはオブジェクトをまたはクリーンアップします。りてをするとオブジェクトのメモリはそのでりてられます `cpython` はのタイプのオブジェクトをすることがあります。ほとんどの、きオブジェクトプールをせず、なときにりてをします。に、`refcount` が0にデクリメントされると、GCはそれをクリーンアップします。

ガベージコレクション

1960にJohn McCarthyは、Lispでされている `refcounting` アルゴリズムをしたときに、ガベージコレクションをカウントするになをしました2つのオブジェクトがでおいをするとどうなりますからがいつもおいををする、がなくとも、どうしてこれら2つのオブジェクトをガベージコレクションすることができますかこのは、リングバッファまたはリンクリストのの2つのしたエントリなど、のデータにもぶ。Pythonはこのを、ガベージコレクションとばれるのがベージコレクションアルゴリズムでいようにしようとします。

に、Pythonでオブジェクトをするときには、そのオブジェクトをリンクリストのにします。にはPythonがこのリストをループして、リストのオブジェクトがどのオブジェクトをしているかをチェックし、それらがリストにもまれているかどうかをチェックしますなぜそれらがでないのかかります。では、がいつくかをめるヒューリスティックがありますが、をにするためののコレクションのにあるとしようとしします。0よりきい `refcount` をつものは、"Generation 1" これは、すべてのオブジェクトがに0のリストにあるわけではないためです、このループはあまりにされません。これは、ガベージコレクションがるです。デフォルトでは3つのがPythonにありますオブジェクトの3つのリンクされたリスト。のリスト0にはすべてのしいオブジェクトがまれます。GCサイクルがしてオブジェクトがされない、それらは2のリスト1にし、2のリストでGCサイクルがし、それらがまだされない、3のリスト2。3のリスト「2」とばれているのは、ゼロインデックスなのでは、の2つよりもはるかにないでガベージコレクションされます。オブジェクトがきするとGCedになるはく、アプリケーションのにGCされるので、1のGCごとにをすることはありません。さらに、ほとんどのオブジェクトはくガベージ・コレクションされることがわかりました。これからは、いうちにくるので、これらの「いもの」とぶことにします。これは「」とばれ、60にめてされたものです。

にの2となり、の3のリストはなスケジュールでガベージコレクションされません。のいオブジェクト3のリストにあるものの、にはGCサイクルをまだっていないオブジェクトと、リストのオブジェクトののが25をえたときにチェックされます。これは、3のリストはであるためにはのリストにされることはないため、にガベージコレクションされたときにのみえてしまいます、のオブジェクトをたくさんするアプリケーションの、GCサイクル3のリストにはかなりいがかかことがあります。をすることによって、「オブジェクトのにおけるされた」をする。リストがくなればなるほどGCはくなりますが、GCをするはなくなりますここでは、Martin VonLöwisによるこのヒューリスティックののがきます。3または「した」リストでガベージコレクションをするは、「ガベージコレクション」とばれます。

だからガベージコレクションは、GCをとしそうもないオブジェクトをスキャンするがないため、をしますが、をるのにどのようにちますかおそらくそれほどうまくいかないかもしれません。これらのサイクルをにるはのようになります。

```
/* 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)
```

ガベージコレクションがこれをけるは、リストのさを々のカウントとしてできることです。しいオブジェクトをにするたびに、このカウントをやし、オブジェクトをのにするか、またはオブジェクトをのにするか、それをデアロックします。には、GCサイクルのわりに、このカウントの2のはに0にするがあります。そうでないは、リストのかがのらかのであり、りくことができます。しかし、もうつのがあります。ったオブジェクトにPythonのメソッド__del__があるとどうなりますか __del__ は、Pythonオブジェクトがされる__del__ にびされます。で2つのオブジェクトがっているは、__del__ メソッドを、たちは1をすると、のをさないことをすることはできません__del__ メソッドを。されたとして、のようにいたとしてみてください。

```
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)
```

AのインスタンスとBのインスタンスをいにポイントするようにし、それらはじガベージコレクションサイクルにわるでしようかたちがランダムに1つをし、にAのインスタンスをdeallocしたとしましょう。Aの__del__メソッドがびされ、それがされ、Aがされます。にBにて、__del__メソッドとんで、おっと Segfault Aはしません。このをするには、まず__del__メソッドをして__del__、にすべてをににするのパスをびすことでこれをするすることができます。しかし、これはのをします1

つのオブジェクト`__del__`メソッドがGCedしようとしているのどこかでたちをしていますかまだサイクルがありますが、にはどちらのオブジェクトもGCがされていなくてもにGCすることはできません。オブジェクトがデータでなくても、その`__del__`メソッドでできることにしてください。Pythonはこれをチェックし、`__del__`メソッドがひかれたにオブジェクトrefcountがしたにGCingをします。

これをCPythonでするには、`__del__`なゴミのグローバルリストにそれらのGC `__del__`なオブジェクトらかののと`__del__`メソッドをつものをりけて、それをにすことです。

```
/* list of uncollectable objects */
static PyObject *garbage = NULL;
```

Examples

カウント

Pythonのメモリのは、カウントでされます。

オブジェクトがされるたびにえ、にされる、カウントはにします。されるとえ、がになる、カウントはにします。

カウントがゼロになると、オブジェクトはちにされ、メモリはただちにされます。したがって、ほとんどの、ガベージコレクタはない。

```
>>> 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
```

のをさらにするために

```
>>> 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

```

サイクルガベージコレクタ

ガベージコレクタがなのは、サイクルがあるです。サイクルのシンプルは、AがBをし、BがAをし、AまたはBのいずれもしないものです。AまたはBのいずれもプログラムのどこからでもアクセスできないため、にできます。それらのカウントは1であるため、カウントアルゴリズムだけではすることはできません。

```

>>> 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

```

サイクルはにくすることができます。もしAがBをしてCをしているならば、AをすZをしているならば、ガーベージコレクションフェーズまではAからZのいずれもされません。

```

>>> 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
```

del コマンドの

`del v` をしてスコープからをするか、または `del v[item]` または `del[i:j]` をしてコレクションからオブジェクトをするか、または `del v.name` をしてをするか、オブジェクトは、デストラクティブにされたメモリをトリガしません。オブジェクトは、カウントが0になったときにのみされます。

```
>>> 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
```

プリミティブオブジェクトの

アプリケーションのについては、プリミティブがにはフードのでもされることです。をてみましょう。-5と256ののすべてのにして、Pythonはにじオブジェクトをします

```
>>> import sys
>>> sys.getrefcount(1)
797
>>> a = 1
>>> b = 1
>>> sys.getrefcount(1)
799
```

`refcount`がすることにしてください。つまり、`a`と`b`は1プリミティブをするときじオブジェクトをします。しかし、よりきいの、Pythonはにはとなるオブジェクトをしません。

```
>>> a = 999999999
>>> sys.getrefcount(999999999)
3
>>> b = 999999999
```

```
>>> sys.getrefcount(999999999)
3
```

999999999のrefcountはaとbにするときにされないため、2つのなるオブジェクトをすることができません。ただし、にじプリミティブがりてられています。

オブジェクトのをする

```
>>> import sys
>>> a = object()
>>> sys.getrefcount(a)
2
>>> b = a
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

にオブジェクトのりてをする

Python 2と3のでrefcountが0でなくても、オブジェクトのりてをにすることができます。

どちらのバージョンもctypesモジュールをしています。

これをうと、Pythonがになり、トレースバックなしでクラッシュするがあります。このをすると、セキュリティのがするがありますまったくえられません。しないことがなオブジェクトのみをりてしてください。これまで

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
```

、りてされているオブジェクトへのがあると、Pythonはのをきこすか、トレースバックなしでクラッシュします。ガベージコレクタがそのオブジェクトをしなかったはおそらくありました...

Noneをりてすると、なメッセージがされます。Fatal Python error: deallocating None。クラッシュするにFatal Python error: deallocating Noneします。

ガベージコレクションの

メモリクリーンアップがされるときにをえる2つのアプローチがあります。それらは、プロセスが

されるとでクリーンアップをトリガーするにします。

ガベージコレクタは、コレクタがするにをえるコレクタしきいをすることによってできます。Pythonは、のメモリシステムをしています。しいオブジェクトはのにされます - 0およびきったコレクションでは、オブジェクトはいにします。のにした- **generation2**を、らがされなくなりしました。

しきいはのスニペットをしてできます。

```
import gc
gc.set_threshold(1000, 100, 10) # Values are just for demonstration purpose
```

のは、ののためのしきいをします。りりのがりりのを1000をえるたびに、ガベージコレクターがびされます。

プロセスをするために、いはにはクリーニングされません。2と3のはオプションで、いのクリーニングをします。 **generation0**が **generation1**をせずに100をしたには、 **generation1**がされます。に、 **generation2**のオブジェクトは **generation1**におけるものが **generation2**にれることなく10したにのみされます。

しきいをでするのがなのは、プログラムがりてをせずにのさなオブジェクトをりてて、ガベージコレクタがにされる **generation0_threshold**オブジェクトのりてです。コレクターはにですが、なのオブジェクトでする、パフォーマンスのがします。とにかく、しきいをするためのすべてのにするサイズはなく、ユースケースはできるものです。

でコレクションをトリガするには、のスニペットのようにします。

```
import gc
gc.collect()
```

ガーベッジ・コレクションは、またはなメモリーではなく、りりおよびりりのについてにトリガーされます。したがって、きなオブジェクトをしてする、クリーンアップがトリガーされるにメモリがなくなるがあります。これはガベージコレクタをでびすためのいユースケースです。

それはですが、それはされたではありません。メモリリークをけることがのです。とにかく、きなプロジェクトではメモリリークをすることができですが、ガベージコレクションをでトリガすることは、さらにデバッグするまではなとしてできます。

プログラムの、ガベージコレクションはベースまたはイベントベースでトリガーできます。のは、のののにをトリガーするWebサーバーです。では、のタイプのがされたときにガベージコレクションをトリガーするWebサーバー。

ガベージコレクションがクリーンアップするのをつはありません

ガベージコレクションがクリーンアップされるというのは、ガベージコレクションサイクルがクリーンアップするのをつべきではありません。

に、ガベージコレクションがファイルハンドル、データベース、およびオープンネットワークを
じるのをすべきではありません。

えば

のコードでは、`f`がファイルへのものであった、ファイルがのガベージコレクションサイクルでじら
れることをとしています。

```
>>> f = open("test.txt")
>>> del f
```

もっとなは、`f.close()`をびすこと`f.close()`。[コンテキストマネージャ](#)ともばれる`with`ステート
メントをすることで、よりされたをうことができます。

```
>>> with open("test.txt") as f:
...     pass
...     # do something with f
>>> #now the f object still exists, but it is closed
```

`with`ステートメントをすると、いているファイルのにコードをインデントすることができます。
これにより、ファイルのオープンをににできます。`while`ブロックでがしても、にファイルをクロ
ーズします。

オンラインでガベージコレクションをむ <https://riptutorial.com/ja/python/topic/2532>/ガベージコレ
クション

85: キューモジュール

き

キューモジュールは、マルチプロデューサ、マルチコンシューマキューをします。これは、がの
スレッドでにされるがあるに、スレッドプログラミングでにちます。 1.キュー2. LifoQueue 3.
PriorityQueueるのある1.フルキューオーバーフロー2.キューアンダーフロー

Examples

な

```
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))
```

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

オンラインでキューモジュールをむ <https://riptutorial.com/ja/python/topic/8339/キューモジュール>

86: クライアントとサーバーのソケットとメッセージの

き

はセキュリティのでされます。IDEAMODE CTRをするPythonの/のはそれほどくありません。
このドキュメントの

ステーションにおけるRSAデジタルのとSHA-1であるメッセージののためにハッシングをする。
シンプルなキートランスポートプロトコルをします。IDEAをしてをする。ブロックのモードは
カウンタモードです

Python 2.7ダウンロードリンク <https://www.python.org/downloads/>

ライブラリ

* **PyCrypto** ダウンロードリンク <https://pypi.python.org/pypi/pycrypto>

* **PyCryptoPlus** ダウンロードリンク <https://github.com/doegox/python-cryptoplus>

ライブラリのインストール

PyCrypto ファイルをします。ディレクトリにして、linuxalt + ctrl + tのターミナルをき、CMDshift
+クリック+コマンドプロンプトをここできますをきます。その、python setup.py installWindows
OSでPythonがしくされていることをする

PyCryptoPlusのライブラリとじです。

タスクのタスクは2つのにかれています。1つはハンドシェイクプロセスであり、もう1つはプロ
セスです。ソケット

- とのとのハッシュとして、すぐソケットをするがあります。ソケットをするには、"import
socket"をつのモジュールをインポートし、IPアドレスとポートをユーザーからするソケッ
トとクライアントまたはバインドサーバーするがあります。

..... クライアント

```
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))
```

..... サーバ

```
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**"は**accept**とメッセージングのをできるようにします。わりに、"**socket.AF_INET**、**socket.SOCK_DGRAM**"もできますが、そのには**setblockingvalue**をしなければなりません。

ハンドシェイクプロセス

- CLIENTまず、とをします。とをするには、いくつかのモジュールをインポートする必要があります。それらはのとおりです。Crypto import RandomおよびCrypto.PublicKey import RSAから。キーをするには、なコードをくがあります。

```
random_generator = Random.new().read
key = RSA.generate(1024, random_generator)
public = key.publickey().exportKey()
```

random_generatorは、"**Crypto import Random**"モジュールからしたものです。キーは、"**Crypto.PublicKey import RSA**"からし、ランダムなをすることによって1024のをします。Publicは、にされたからをエクスポートしています。

- CLIENTとをしたら、をハッシュしてSHA-1ハッシュをしてサーバにするがあります。SHA-1ハッシュをするには、"**import hashlib**"をくことによるのモジュールをインポートする必要があります。をハッシュするには、2のコードをきます

```
hash_object = hashlib.shal(public)
hex_digest = hash_object.hexdigest()
```

ここで、hash_objectとhex_digestはです。その、クライアントはhex_digestとpublicをサーバにし、サーバはクライアントからしたハッシュとキーのしいハッシュをしてサーバをします。しいハッシュとクライアントからのハッシュがする、のにります。クライアントからされたパブリックはであるため、サーバでキーとしてすることはできません。これをし、をrsaにするには、server_public_key = RSA.importKey(getpbk)とくがあります。ここでgetpbkはクライアントからのです。

- SERVERのステップは、セッションキーをすることです。ここでは、"os"モジュールをしてランダムキー "key = os.urandom16"をしました。このキーは16ビットのさのキーをえ、その、そのキーを "AES.MODE_CTR"でし、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.shal(encrypto)
```

```
en_digest = en_object.hexdigest()
```

したがって、en_digestがセッションキーになります。

- SERVERハンドシェイクののは、クライアントからしたとサーバーでしたセッションをすることです。

```
#encrypting session key and public key  
E = server_public_key.encrypt(encrypto,16)
```

の、サーバーはとしてクライアントにキーをします。

- CLIENTサーバからとセッションのされたをした、クライアントはにとにしたをしてそれらをします。されたとセッションがであるため、evalをしてとしてすがあります。がした、がじをしていることをすると、ハンドシェイクもする。するには

```
en = eval(msg)  
decrypt = key.decrypt(en)  
# hashing sha1  
en_object = hashlib.sha1(decrypt) en_digest = en_object.hexdigest()
```

はここでSHA-1をして、でむことができるようにしました。

プロセス

プロセスでは、IDEAMODE_CTRのKEYとしてのセッションをするがあります。は、セッションキーをしてIDEA.MODE_CTRでメッセージをおよびします。

- IDEAののためには、16bitのとびしでなければならないカウンタがです。MODE_CTRではカウンタがです。たちがしハッシュしたセッションのサイズは、IDEAのキーをえる40になりました。したがって、セッションキーのサイズをさくするがあります。のために、の[valuevalue]でされたのpythonをできます。は、ユーザーののにじてのにできます。たちの、キーから016のをる「キー[16]」をしました。このは、キー[1:17]やキー[16]のようにくのでうことができます。のは、のために3つのをとるIDEA.newをくことによってしいIDEAをすることです。のはKEY、2のはIDEAこのはIDEA.MODE_CTRのモードになり、3のはcounter=でびしなになります。カウンタ=は、によってされるのサイズをします。counter=をするには、なをするがあります。この、はラムダをすることによってKEYのサイズをしました。ラムダをするわりに、counter = Utilをしてcounter =にしてランダムなをすることができます。Counter.Utilをするには、からカウンタモジュールをインポートするがあります。したがって、コードはのようになります。

```
ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
```

IDEAとして "ideaEncrypt"をしたら、のをしてすべてのメッセージをできます。

```
eMsg = ideaEncrypt.encrypt(whole)
```

```
#converting the encrypted message to HEXADECIMAL to readable eMsg =
eMsg.encode("hex").upper()
```

このコードセグメントでは、wholeはされるメッセージであり、eMsgはされたメッセージです。メッセージをした、はそれをHEXADECIMALにしてみりにし、upperはをにするみみです。その、このされたメッセージはのためににられる。

•

されたメッセージをするには、じとじキーをしてのをするがありますが、はがされたメッセージをします。とじコードです。しかし、メッセージをするに、16からメッセージをするがあります。というのも、たちのでは、されたメッセージを16でコードしてみやすくするからです。したがって、コードはのようになります。

```
decoded = newmess.decode("hex")
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
dMsg = ideaDecrypt.decrypt(decoded)
```

これらのプロセスは、とのためにサーバーとクライアントのでされます。

Examples

サーバーの

```
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.shal(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 shal
    en_object = hashlib.shal(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 HEXADECEIMAL to decrypt the ecrpyted 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")

```

クライアントの

```

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.shal(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()

```

オンラインでクライアントとサーバーのソケットとメッセージの/をむ

<https://riptutorial.com/ja/python/topic/8710/クライアントとサーバーのソケットとメッセージの->

87: クラス

き

Pythonはなスクリプティングとしてだけでなく、オブジェクトプログラミングパラダイムもサポートしています。クラスはデータをし、そのデータをするメソッドをします。これらのメソッドはすべてのオブジェクトのにあります。さらに、クラスでは、なのをデータのなからすることによって、がです。

クラスをするコードは、にみやすく、し、するのがよりです。

Examples

な

Pythonのは、Java、C++などのオブジェクトでされているのえについています。しいクラスは、のクラスからのようにすることができます。

```
class BaseClass(object):  
    pass  
  
class DerivedClass(BaseClass):  
    pass
```

BaseClassはすでにする クラスで、DerivedClassはBaseClassからをするまたはサブクラスするしいクラスです。 Python 2.2、すべてのクラスはすべてのみみのクラスであるobjectクラスをにしていobject。

のではRectangleクラスをしています。これはにobjectからしてい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)
```

Rectangleクラスは、Squareクラスをするためのクラスとしてできます。はのなケースです。

```
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
```

SquareクラスはにRectangleクラスとオブジェクトクラスのすべてのをします。 `super()` はRectangleクラスの`__init__()` メソッドをびすためにされ、クラスのオーバーライドされたメソッドをにびします。 Python 3では、 `super()` はありません。

クラスオブジェクトは、クラスのにアクセスしてできます。

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

をみる

`issubclass(DerivedClass, BaseClass)` す `True` `DerivedClass` のサブクラスである `BaseClass`

`isinstance(s, Class)` す `True` `s` はのインスタンスであるは `Class` またはクラスの `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
```

クラスとインスタンス

インスタンスはインスタンスごとにであり、クラスはすべてのインスタンスでされます。

```
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

```

クラスはこのクラスのインスタンスでアクセスできますが、`class`にりてると、クラスをシャドウするインスタンスがされます

```

c2.x = 4
c2.x
# 4
C.x
# 2

```

インスタンスからクラスをさせると、いくつかのせぬをくことにしてください。

```

class D:
    x = []
    def __init__(self, item):
        self.x.append(item) # note that this is not an assignment!

d1 = D(1)
d2 = D(2)

d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]

```

バインドされた、バインドされていない、およびメソッド

バインドされたメソッドとバインドされていないメソッドのアイデアは、[Python 3ではされました](#)。Python 3では、クラスでメソッドをするときに、`def` キーワードをしてオブジェクトをしています。これはので、のクラスはそのとしてします。のでは、メソッド_fをクラス_Aでし、_{Af}になります。

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)
```

Python 2では、がなりました。クラスのオブジェクトは、のクラスインスタンスにバインドされていないため、アンバインドされたメソッドと呼ばれる`instancemethod`のオブジェクトににきえられました。 `.__func__` プロパティをしてとなるにアクセスすることは `.__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 ...>
```

のはによってされます。メソッドはPython 3のとしてされますが、Python 2ではされています。

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
```

のバージョンのPython/メソッドでは、クラス`A`インスタンスをのとしてすと、`A.f`をびすことができます。

```
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
```

としたクラスのインスタンスである`a`がであるか、`a.f`、そのまあ、にこれはじでなければなりません`f`クラスの`A`のみ、それはらかのでそれがオブジェクトにされたことを「る」がある、とし

た Python でこれがびされるメソッド-。 a a

のようにザラザラのはのとおりです。きみ_{af}びす `__getattr__` かどうかをにチェック、というのがあり `a.a.f`、クラスのチェックそうでない、`A`、それは、そのようなをつメソッドがまれているかどうかをそれはありません、およびしいオブジェクト `m` の `method` をする `Af` における `m.__func__`、およびオブジェクトへの `a` における `m.__self__`。このオブジェクトがとしてびされると、`m(...)` $\Rightarrow$ `m.__func__(m.__self__, ...)` ます。このため、このオブジェクトはバインドされたメソッドとされます。これは、びされると、バインドされたオブジェクトにのとしてすることができするためです。これらは Python 2 と 3 でじようにします。

```
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
```

に、Python にはクラスメソッドとメソッド、つまりなのメソッドがあります。クラスメソッドは、のメソッドとじようにします。ただし、オブジェクトにしてびされると、オブジェクトではなくオブジェクトのクラスにバインドされます。したがって、`m.__self__ = type(a)`。このようなバインドされたメソッドをびすと、`a` のクラスがのとしてされます。メソッドはさらにです。もバインドしませんし、なしでをにします。

```
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
```

クラスメソッドは、インスタンスでアクセスされたでもクラスにバインドされていることにしてください。


```

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

```

レベルでは、メソッド、`__get__`メソッドなどは、には`__set__`、`__del__`メソッドをびすであることに`__get__`して`__set`。 `classmethods`と`staticmethods`のについては

- Pythonの`@staticmethod`と`@classmethod`のいはいですか
- のための`@classmethod`と`@staticmethod`の

しいスタイルといスタイルのクラス

Python 2.x 2.2.0

しいスタイルのクラスがPython 2.2でされ、クラスとをしました。それらはトップレベルの`object`からし`object`。しいスタイルのクラスはユーザーで、みみとによくています。

```

# 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

```

いスタイルのクラスは`object`からしません。のインスタンスは、にみみの`instance`でされ`instance`。

```

# 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

Python 3では、いスタイルのクラスがされました。

Python 3のしいスタイルのクラスはに `object` からされるので、 `MyClass(object)` もうするはありません。

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

インスタンスのデフォルト

にえばのがまれているは、このようにデフォルトをすることができます

```
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
```

コンストラクタのリストなどのオブジェクトをするときはがです。のをえてみましょう。

```
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
```

このは、Pythonのデフォルトのパラメータがのにバインドされ、のではバインドされないというによってします。インスタンスでされないデフォルトのインスタンスをするには、のようなをす

るがあります。

```
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
```

[Mutable Default Arguments](#)と「[Least Astonishment](#)」およびなデフォルトをしてください。

PythonはC3アルゴリズムをして、メソッドをむクラスをするをします。これは、Method Resolution OrderMROとされます。

ここになががあります

```
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'
```

ここでFooBarをインスタンスすると、fooをべると、Fooのがにつかることがわかります

```
fb = FooBar()
```

そして

```
>>> fb.foo
'attr foo of Foo'
```

FooBarのMROはのとおりです。

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

PythonのMROアルゴリズムは

1. さはにえはFooBar、 Foo
2. object によってブロックされているBar と
- 3.

なはされません。

つまり、たとえば、BarはFooBarからできませんが、FooBarはBarからします。

Pythonのなについては、[wikipediaのエントリー](#)をしてください。

のもうつのなは`super`です。 `super`はクラスのフィーチャーをフェッチできます。

```
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()
```

クラスの`init`メソッドをつた。すべてのクラスにの`init`メソッドがある、の`inheritance`をしてから、`init`メソッドがにするクラスをびすだけです。

たとえばののFooクラスの`init`メソッドがびさバークラスがびされていない`init`の

```
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()
```

```
foobar init
foo init
```

しかし、**Bar**クラスがされているわけではありません。な**FooBar**クラスのインスタンスは、**Bar**クラスと**Foo**クラスのインスタンスでもあります。

```
print isinstance(a, FooBar)
print isinstance(a, Foo)
print isinstance(a, Bar)
```

```
True
True
True
```

とドットき

は、はクラスのとそれは、のっているオブジェクトである `__get__`、`__set__`、または `__delete__` なを。

データ `__set__` には `__set__` または `__delete__` いずれかがあり `__delete__`

これらは、インスタンスのルックアップをすることができ、`staticmethod`、`classmethod`、および `property` をするためにされ `property`。のルックアップえば、`Foo` クラスのインスタンス `foo` は、`bar` してい `bar` - つまり `foo.bar` はのアルゴリズムをします

1. `bar` はクラス `Foo` でされます。それはそこにあり、それがデータであるには、データがされています。これは `property` がインスタンスのデータへのアクセスをするであり、インスタンスはこれをオーバーライドできません。データがしないは、
2. `bar` はインスタンス `__dict__` でされます。このため、インスタンスからびされるメソッドをのルックアップでオーバーライドまたはブロックすることができます。インスタンスに `bar` がするは、そのインスタンスがされます。そうでなければ、々は
3. クラス `Foo` で `bar` し `bar`。 **Descriptor** の、プロトコルがされます。これは、このコンテキストでは、バインドされていないメソッド、`classmethod`、および `staticmethod` がどのようにされるかです。それのは、にそのオブジェクトをします。そうでないは、`AttributeError`

クラスメソッド

クラスメソッドは、クラスのインスタンスをするのをします。するために、をてみましょう。

な `Person` クラスがあるとしましょう

```
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 + ".")
```

このクラスのインスタンスをして、とのを々にするがあればかもしれません。これをう1つののは、`last_name` にオプションのパラメータをすることです。されていないはをします

```
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 + ".")

```

しかし、このコードには2つのきながあります。

1. パラメータの `first_name` と `last_name` あなたがなをすることができるので、らわしいです `first_name`。また、こののをつよりくのケースやパラメータがあれば、`if / elif / else` ブランチングはなものになります。
2. それほどではありませんが、それでもするがありますもし `last_name` が `None` であれば、`first_name` はスペースで2つのものにされませんかのやのがあります...

クラスメソッドをします。のイニシャライザを `from_full_name` するのでは `from_full_name`、`from_full_name` というのイニシャライザをし、みみの `classmethod` デコレータでデコレートします。

```

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 + ".")

```

`from_full_name` のとして `self` わりに `cls` を `from_full_name` ます。クラスメソッドは、えられたクラスのインスタンスではなく、のクラスにされますこれは、`self` がするものです。したがって、`cls` が `Person` クラスである、`from_full_name` クラスメソッドからされるは `Person(first_name, last_name, age)` であり、`Person` クラスのインスタンスをするために `Person` の `__init__` をします。に、`Employee` of `Person` サブクラスをする、`from_full_name` は `Employee` クラスでもします。

これがどおりにすることをすために、`__init__` ししないでので `Person` インスタンスをしましょう

```

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.
```

そのの

- [のためのPython @classmethodと@staticmethod](#)
- <https://docs.python.org/2/library/functions.html#classmethod>
- <https://docs.python.org/3.5/library/functions.html#classmethod>

クラス

クラスは、オブジェクトのなをにする。このでは、々はにするにんでいます。によって、にむすべてののにアクセスできます。

```
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
```

モンキーパッチ

この、"monkey patching"とは、されたクラスにしいまたはメソッドをすることをします。たとえば、クラス`A`をのようにしたとします。

```
class A(object):  
    def __init__(self, num):  
        self.num = num  
  
    def __add__(self, other):  
        return A(self.num + other.num)
```

しかし、はコードののにのをしたいとっています。このがのようになっているとします。

```
def get_num(self):  
    return self.num
```

しかし、これを`A`メソッドとしてどのようにしますかそれはに、をで`A`にするだけです。

```
A.get_num = get_num
```

なぜこれはしますかはのオブジェクトとまったくじオブジェクトなので、メソッドはそのクラスにします。

`get_num`は、すべてののにされているものと、`A`しいインスタンス

これらのは、そのクラスまたはそのサブクラスのすべてのインスタンスでにできます。えば

```
foo = A(42)  
  
A.get_num = get_num  
  
bar = A(6);  
  
foo.get_num() # 42  
  
bar.get_num() # 6
```

のとはなり、このテクニックはのみではしません。また、それはスタイルとはなされません。

すべてのクラスメンバーをリストする

`dir()`は、クラスのメンバーのリストをするためにできます。

```
dir(Class)
```


例えば

```
>>> 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']
```

「」のメンバーだけをのるのがです。これは、__で__ないのメンバーをリストするなをしてうことができます。

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

クラスは__dir__()メソッドをすることができます。そのメソッドがする、dir()は__dir__() dir()をびします。そうでないと、Pythonはそのクラスのメンバのリストをしようとします。これは、dirがしないをもたらすがあることをします。 [のPythonドキュメント](#)からな2つの

オブジェクトが**dir** をしない、は、オブジェクトの**dict**されているとオブジェクトからをするようにのをします。としてられるリストはずしもなものではなく、オブジェクトがカスタム**getattr**をつときはかもしれません。

dirは、にプロンプトでのののためにされているため、にまたはしてされたセットをしようとするよりいいセットをしようとします。リリース。たとえば、がクラスの、メタクラスはリストにありません。

クラスの

クラスは、のオブジェクトのなをするテンプレートとしてします。ここにがあります

```
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))
```

のをと、すべきがいくつかあります。

1. クラスはデータとメソッドでされています。
2. およびメソッドは、にのおよびとしてされます。
3. するdocstringにかけられているように、`__init__()`メソッドはイニシャライザとされます。のオブジェクトのコンストラクタにし、しいオブジェクトやクラスのしいインスタンスをするときににされるメソッドです。
4. クラスにされるがにされ、クラスとされます。
5. クラスオブジェクトののインスタンスにされるはインスタンスとされます。それらはに`__init__()`でされます。これはではありませんが、`__init__()`でされたは、されるにアクセスされるリスクがあります。
6. クラスにまれるすべてのメソッドは、そのオブジェクトをのパラメータとしてします。`self`というはこのパラメタにわれます。`self`にはによるものですが、`self`というはPythonではなをちませんが、これはPythonのもされるの1つです。
7. のでのオブジェクトプログラミングにされているものは、いくつかのでくかもしれません。ひとつは、Pythonには`private`のがないということです。そのため、デフォルトではすべてがC++/Java `public`キーワードのをしています。については、このページの「プライベートクラスメンバー」のをしてください。
8. クラスのメソッドのには、`__functionname__(self, other_stuff)`というのがあります。そのようなメソッドはすべて「マジックメソッド」とされ、Pythonのクラスのなです。たとえば、Pythonでのオーバーロードは、のでされます。については、[を](#)してください。

Personクラスのインスタンスをいくつかってみましょう

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

、3つのPersonオブジェクト、`kelly`、`joseph`、および`john_doe`ます。

ドットをして、インスタンスからクラスのにアクセスできます。ここでもクラスとインスタンスのいにしてください。

```
>>> # Attributes
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

じドットを使ってクラスのメソッドをすることができ.

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>> print(john_doe)
'John Doe'
>>> john_doe.rename("John")
'Now my name is John'
```

プロパティ

Pythonクラスは、のオブジェクトのようにえるプロパティをサポートしますが、カスタムビヘイビアとドキュメントをするがあります。

```
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
```

オブジェクトのクラス`MyClass`は、プロパティ`.string`をっているようにえますが、はにされています。

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

のようななとに、プロパティでは、バリデーションやそののをそれらのにすることができます。これは、パブリックAPIユーザにレベルのヘルプをするがあるににです。

プロパティののなは、クラスがにはされず、されたときにのみされるである"をできるようにすることです。

```
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
```

シングルトンクラス

シングルトンは、クラスのインスタンスを1つのインスタンス/オブジェクトにするパターンです。
pythonのシングルトンデザインパターンのについては、[こちらをしてください](#)。

```
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
```

のはあなたのクラスをることです。この[え](#)のにて、シングルトンクラスをします。

```
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)
```

するには `Instance` メソッドをできます

```
@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
```

オンラインでクラスをむ <https://riptutorial.com/ja/python/topic/419/クラス>

88: クラスインスタンスの `__str__` および `__repr__` メソッド

のメソッドにする

のがされているは、それがっている、ややです `__str__` にしいえば、「スペースのエース」をし、メソッド `__repr__` す `eval` やさしいを。

、 `repr()` Pythonドキュメントはこれをにメモしています

くのでは、このは `eval` にされるときにじをつオブジェクトをするをそうとします。さ
もなければ、そのはオブジェクトののをむでまれたですくの、オブジェクトのとアド
レスをむがあります。

つまり、 `__str__` して `__str__` ように「Ace of Spades」のようなものをすことができますが、わり
に `__repr__` して `Card('Spades', 1)`

このは、ある「」で `eval` にすことができます

```
object -> string -> object
```

このようなメソッドのはのとおりです。

```
def __repr__(self):  
    return "Card(%s, %d)" % (self.suit, self.pips)
```

ノート

[1]このはのものです。されるはcpythonのものです。

[2]あなたはすでにこの `str()` / `repr()` のをかっていて、かっていないかもしれません。バックスラ
ッシュなどのをむが `str()` をしてにされると、バックスラッシュはそのままされ `str()` されます。そ
れらが `repr()` をしてにされるとたとえば、されるリストのとして、バックスラッシュはエスケ
ープされて2されます。

Examples

だから、ちょうどあなたののクラスをPythonでしました。これはトランプカードをカプセルした
きちんとしたさなクラスです

```
class Card:
```

```
def __init__(self, suit, pips):
    self.suit = suit
    self.pips = pips
```

コードのどこかで、このクラスのいくつかのインスタンスをします。

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs', 4)
six_of_hearts = Card('Hearts', 6)
```

あなたは "" をすためにカードのリストをしました。

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

、デバッグに、あなたののをたいので、にてをするのかをします

```
print(my_hand)
```

しかし、あなたがってくるものは、ぎこちないです。

```
[<__main__.Card instance at 0x0000000002533788>,
 <__main__.Card instance at 0x00000000025B95C8>,
 <__main__.Card instance at 0x00000000025FF508>]
```

していると、1のカードだけをしようとしてします

```
print(ace_of_spades)
```

そしてもう、あなたはこのなをます

```
<__main__.Card instance at 0x0000000002533788>
```

もくない。これをすつもりです。

まず、ここでがこっているのかをすることがです。あなたが `print(ace_of_spades)` をいたとき、Pythonにあなたのコードが `ace_of_spades` びしている `Card` インスタンスにするをしたかったと `ace_of_spades` 。そして、であるために、それはしました。

そのは、オブジェクトの `type` とオブジェクトの `id` の2つのなビットでされます。2のだけ16は、`print` びしにオブジェクトをにするのにです。[1]

にいたのは、Pythonにオブジェクトのを "でえ" て、それをあなたにするようにんだことでした。じマシンのよりなバージョンは、のようなものです。

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```


のでは、`Card`インスタンスをにし、にインスタンスをします。

あなたがしているは、あなたがPythonに`Card`をするための`Card`クラスについてっておくがあるすべてのことについて、あなたが`Card`インスタンスをにするをえていないというのためにこります。

そしてそれがわからないので、のうちにあなたが`str(ace_of_spades)`いたとき、それはあなたがたもの、つまり`Card`インスタンスのなをえました。

その1

しかし、カスタムクラスのインスタンスをどのようににするかをPythonにえることができます。そして、たちがこれをうは、`__str__` "dunder"ダブルアンダースコアのまたは "magic"メソッドのです。

クラスインスタンスからをするようにPythonにするたびに、クラスで`__str__`メソッドがされ、ひされます。

たちの`Card`クラスののアップデートバージョンをえてみましょう。

```
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)
```

ここでは、`__str__`メソッドを`Card`クラスにしました。これは、なの、フォーマットされたをします。

をすことのをするために、`""`はここではでしてはいますが、にするのではなく、するとわれるかもしれませんが、`str(ace_of_spades)`、あなたのメインプログラムで`print`のびしをしなくても、`__str__`がをすことをして`__str__`。

`__str__`メソッドはメソッドなので、のは`self`あり、`__str__`をけることもすべきでもありません。

カードをよりユーザーフレンドリーなですというにり、ひすると

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

たちのがはるかにれていることがわかります

```
Ace of Spades
```

な、たちはですよ

たちのをカバーするだけで、したのをしたことをもうして、`Card`インスタンスのリストである`hand`してみましょう。

そこでのコードをします

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

そして、たちのくべきことに、それらのい16コードをもうます

```
[<__main__.Card instance at 0x00000000026F95C8>,
 <__main__.Card instance at 0x000000000273F4C8>,
 <__main__.Card instance at 0x0000000002732E08>]
```

どうしたのたちは、Pythonに、`Card`インスタンスをするをえてくれました。どうしてらかにれてしまったのでしょうか

その2

まあ、のは、Pythonがリストののをしたいときにはしっています。それは、Pythonがこののために`__str__`をにしないことがしました。

わりに、それはのをします`__repr__`、それがつかからないは、それは「16の」にフォールバックします。[2]

だからはじことをする2つのをらなければならないとっているのですか 1つはのカードとそれがらかのコンテナにっているときのカードを`print`たいときです。

いいえ、に、`__str__`と`__repr__`の`__str__`をするは、クラスがどのようなものになるかをてみましょう

```
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)
```

ここで、つのは `__str__` と `__repr__`、つのはをすることをいて、にじである、(S) によってされるにさ `__str__` び (R) によってされるにされる `__repr__`。

`__str__` メソッドとに、 `__repr__` はをけけず、をします。

それぞれのにどのようながであるかをできます。

```
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)
```

`__str__` メソッドは、 `Card` インスタンスをして `print` ときにびされ、 `__repr__` メソッドは、インスタンスのリストをして `print` するときにびされました。

こので、にったように `str()` をってカスタムクラスインスタンスからをにできるように、 `repr()` というみみをとってクラスのをにすることもできます `repr()`。

えは

```
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)
```

さらに、されていれば、メソッドをびすこともできますただし、でながあります。

```
print(four_of_clubs.__str__())    # 4 of Clubs (S)

print(four_of_clubs.__repr__())   # 4 of Clubs (R)
```

それらについて

Pythonのは、じを `str()` と `repr()` からすようにしたい、メソッドをにするがあります。

そのわりに、そのをするためのみがあります。はこのまであなたをしています。これは、クラスがしていることがし `__repr__` ではなく、 `__str__` を、あなたがするそのクラスのインスタンスをす `str()` またはかどうか、Pythonはあなたにフォールバックします `__repr__` とそれをします。

したがって、にするために、のバージョンのCardクラスをしてください。

```
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)
```

このバージョンは__repr__メソッドのみをしていることにしてください。それにもかかわらず、str()びすとユーザフレンドリーなバージョンになります

```
print(six_of_hearts)           # 6 of Hearts (implicit conversion)
print(str(six_of_hearts))      # 6 of Hearts (explicit conversion)
```

repr()びしとじように

```
print([six_of_hearts])        #[6 of Hearts] (implicit conversion)
print(repr(six_of_hearts))    # 6 of Hearts (explicit conversion)
```

あなたのクラスインスタンスにユーザフレンドリーなで「を」するをえるには、なくともクラスの__repr__メソッドをすることをしてください。メモリがされている、Raymond Hettingerのでは、クラスが__repr__することが、Pythonのコードレビューをうに__repr__ 1つであるとべています。なでデバッグステートメント、クラッシュレポート、またはログファイルにできるは、デフォルトでえられているごろな、そしてしばしばたないタイプ、IDとするとされます。

あなたがのためになるをしたいは、えは、のに、あなたはをすることをおめします__repr__と__str__メソッドを。これらの2つのをではどのようにいけるかについてのはこちら。

のメソッドがされ、eval-round-tripスタイル__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)
```

オンラインでクラスインスタンスの__str__および__repr__メソッドをむ

<https://riptutorial.com/ja/python/topic/4845/クラスインスタンスの---str--および--repr--メソッド>

89: グラフツール

き

Pythonツールを使ってグラフをすることができます

Examples

PyDotPlus

PyDotPlusはGraphvizのDotにPythonインタフェースをするpydotプロジェクトのです。

インストール

のの

```
pip install pydotplus
```

の

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

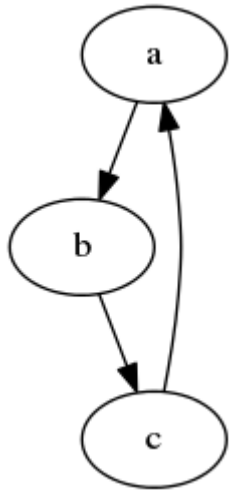
DOTファイルでされたグラフをみむ

- ファイルはDOTであるとされます。みまれ、され、グラフをすDotクラスがされます。たとえば、なdemo.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.
```

のようなsvgScalable Vector Graphicsがられます



PyGraphviz

<http://pypi.python.org/pypi/pygraphviz>の PythonパッケージインデックスからPyGraphvizをしてください

またはをしてインストールします。

```
pip install pygraphviz
```

あなたのオペレーティングシステムとPythonのバージョンに合ったバージョンをつけてインストールしようとしています。

github.comはのものをインストールすることができます

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

<http://pypi.python.org/pypi/pygraphviz>の PythonパッケージインデックスからPyGraphvizをしてください

またはをしてインストールします。

```
easy_install pygraphviz
```

あなたのオペレーティングシステムとPythonのバージョンに合ったバージョンをつけてインストールしようとしています。

DOTファイルでされたグラフをみむ

- ファイルはDOTであるとされます。みまれ、され、グラフをすDotクラスがされます。たとえば、なdemo.dot

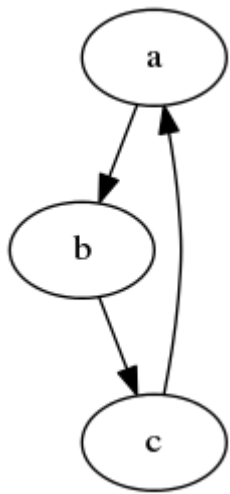
```
digraph demo1 {a -> b -> c; c -> a; }
```

- それをロードしてします。

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
```

```
G.draw('test', format='svg', prog='dot')
```

のようなsvgScalable Vector Graphicsがられます



オンラインでグラフツールをむ <https://riptutorial.com/ja/python/topic/9483/グラフツール>

90: グローバルインタプリタロック GIL をする

なぜ GIL があるのですか

GIL は 1992 に Python スレッドの、CPython をにしてきました。Python コードをするスレッドのをするようにされています。GIL でかかれた Python インタプリタは、のネイティブスレッドが Python バイトコードをにするのをぎます。これにより、プラグインはコードがスレッドセーフであることをにできます。に GIL をロックするだけで、アクティブなスレッドのみをできるため、コードはにスレッドセーフです。

いバージョン GIL は、どれだけのプロセッサとスレッドをとっていても、にできる Python インタプリタのスレッドは 1 つだけです。

これにはいのでくのメリットがありますが、にくののメリットもあります。

GIL は Python のではないことにしてください。したがって、の Python コードから GIL にアクセスすることはできません。Python のすべてののが GIL をするわけではありません。

GIL をつインタプリタ CPython、PyPy、Cython だし、GIL を `nogil` にすることができます

GIL をたないインタプリタ Jython、IronPython

GIL のにする

スレッドがされているとき、スレッドは GIL をロックします。したいスレッドが GIL をし、スレッドがになるまでします。バージョン 3.2 よりの CPython では、のスレッドはの Python をチェックして、のコードがロックをしているかどうかをしますつまり、ロックをしてからしました。このでは、スレッドがするに、ロックをしたスレッドがびスレッドをするため、スレッドのがするがありました。3.2、GIL をしたいスレッドはしばらくのロックをし、そのがすると、のスレッドをにするをします。ただし、これによりがにくなるがあります。については、dabeaz.com ののリンクをしてくださいセクション。

スレッドが I/O をすると、CPython はに GIL をします。イメージライブラリと numpy number crunching オペレーションは、をうに GIL をします。

GIL の

GIL をするの、GIL はです。これは、アプリケーションのをするためにされます。メリット

- GIL がロックされている、ガベージコレクション - スレッドセーフカウントをするがあります。CPython では、garbage コレクションはすべて GIL にびついています。これはきなもの

です。もしGILをしたいのであれば、まだしていなければならないものについては、GILについてのpython.org wikiのをしてください。

- GILをうプログラマーがになる - すべてをロックするのはですが、にコードすることができます
- モジュールののからのインポートをにします。

GILの

GILは、あるスレッドがPythonインタプリタののにpythonコードをできるようにします。つまり、なPythonコードをするプロセスのマルチスレッドはしません。GILにしてスレッドをすると、のスレッドでしたよりもスレッドのパフォーマンスがするがくなります。

<https://wiki.python.org/moin/GlobalInterpreterLock> - それがをすべきかのな、すべてのメリットにする

<http://programmers.stackexchange.com/questions/186889/why-was-python-written-with-the-gil> - にかれた

<http://www.dabeaz.com/python/UnderstandingGIL.pdf> - GILがどのようにし、なぜのコアでくなるのか

<http://www.dabeaz.com/GIL/gilvis/index.html> - GILがスレッドをロックするをすデータの

<http://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/> - GILのをにする

<https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/> - GILのをするの

Examples

マルチプロセッシング。プール

なえは、Pythonでスレッドをうをねるときです「しないでください。プロセスをしてください。」マルチプロセッシングモジュールでは、スレッドのとのでプロセスをできますが、なPoolオブジェクトをすることをおめします。

David BeazleyがにGILにするスレッドのをすためにしたコードをして、マルチプロセッシングをしてきします。プール

デビッド Beazleyのコードは、GILスレッドのをした

```

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

```

マルチプロセッシングをとってきしました.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)

```

スレッドをするわりに、しいプロセスがされます。プロセスはそれのインタプリタであるため、GILのはありません。のでは2つしかありませんが、マシンにコアがあるほどのプロセスをオープンします。のシナリオでは、マシンにプロセッサとじさのさをつようにリストをしたいとえています。プールは、したプロセスのまで、でするようにするをします。がすると、リストのりのがそのプロセスでされます。

は、withステートメントをしwithも、プールをじてしないと、プロセスがしけることがわかりました。リソースをクリーンアップするために、はにプールをじてします。

Cython nogil

CythonはのPythonインタプリタです。GILをしますが、GILをにすることができます。 [ドキュメント](#)をしてください

として、 [David Beazley](#)がにGILにするスレッドのをすためにしたコードをして、nogilをしてきします。

デビッド Beazley のコードは、GIL スレッドのをした

```
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
```

ノギルをとってきしまったサイロンでののみ

```
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
```

あなたがcythonをっているり、それはです。ドキュメントでは、Pythonオブジェクトをしないようにするがあることにしてください。

のにあるコードは、Pythonオブジェクトをしてしてはいけません.GILをにしなければ、Pythonオブジェクトをするものをびさないでください。Cythonはこれをチェックしていません。

オンラインでグローバルインタープリタロックGILをするをむ

<https://riptutorial.com/ja/python/topic/4061/グローバルインタープリタロック-gil-をする>

91: コードブロック、フレーム、および

き

コードブロックは、モジュール、クラス、などとしてできるPythonプログラムのテキストです。いくつかのコードブロックモジュールのようなは1だけされ、のもののようなものはもされます。コードブロックは、のコードブロックをテキストでむことができる。コードブロックは、例えば、をびすびしすることによって、のとしてのコードブロックそのにテキストにまれていてもいなくてもよいをびすことができる。

Examples

コードブロックの

コードブロックタイプ	グローバル	ローカルネームスペース
モジュール	モジュールのns	グローバルとじ
スクリプトファイルまたはコマンド	<code>__main__</code> のためのns	グローバルとじ
コマンド	<code>__main__</code> のためのns	グローバルとじ
クラス	ブロックをむグローバルns	しい
	ブロックをむグローバルns	しい
<code>exec</code> にされる	ブロックをむグローバルns	ブロックをむローカルの
<code>eval()</code> される	びしのグローバルns	のローカルns
<code>execfile()</code> によってみまれるファイル	びしのグローバルns	のローカルns
<code>input()</code> によってみられる	びしのグローバルns	のローカルns

オンラインでコードブロック、フレーム、およびをむ

<https://riptutorial.com/ja/python/topic/10741/コードブロック-フレーム-および>

92: コマンドラインの

き

ほとんどのコマンドラインツールは、にプログラムにされるにしています。をすわりに、これらのプログラムは、データまたはのフラグブールになるがされることをしています。これにより、ユーザーとのプログラムのがPythonファイルをし、にデータをすことができます。このでは、Pythonでのコマンドラインのといについてします。

Examples

argparseのHello world

のプログラムは、ユーザーにこんにちはといます。これは、ユーザーのである1つのをとり、にもえることができます。

```
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!
```

については、[argparseのドキュメント](#)をしてください。

docoptをったな

docoptは、コマンドラインのをします。をするわりに、プログラムのをくだけで、**docopt** はをし、それをとってコマンドラインをします。

```
"""
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)
```

サンプルラン

```
$ 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'}
```

argparseでになをする

2つのをににする `argparse.ArgumentParser.add_mutually_exclusive_group()` をすることができます。のでは、`foo`または`bar`のいずれかがするがありますが、にすることはできません。

```
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
```

--fooと--barのをしてスクリプトをしようとすると、スクリプトはのメッセージをします。

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

argvでのコマンドラインの

Pythonスクリプトがコマンドラインからびされるときはいつでも、ユーザはのコマンドラインをスクリプトにすことができます。これらは、システムからプログラマにできるようになります。sys.argv「argvが」ほとんどのプログラミングでされるなであり、そしてそれは、「umentのVのエクター」をします。

では、sys.argvリストののはPythonスクリプトのですが、りののはスクリプトをびすときにユーザーがすトークンです。

```
# 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']
```

argvのいのもうつのです。スクリプトのがまれているため、にsys.argvのをします。その、りのを1つのにまとめ、に、ログインしているユーザーののにそのをしますチャットプログラムをエミューレートするため。

```
import getpass
import sys

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

のを"で"するときによくされるアルゴリズムは、sys.argvリストをすることsys.argv。1つののは、リストのにして、そのをポップすることです。

```
# 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()

```

argparseをしたカスタムパーサーエラーメッセージ

スクリプトののにじてパーサーエラーメッセージをすることができます。これは

`argparse.ArgumentParser.error`によるものです。のは、とするエラーメッセージスクリプトす
`stderr--foo`えられていないが`--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

```

あなたのスクリプトが`sample.py`とし、する `python sample.py --foo ds_in_fridge`

スクリプトはのをいます

```

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

```

argparse.add_argument_groupによるのなグループ

`argparse.ArgumentParser`をし、`-h`をしてプログラムをすると、ソフトウェアをできるをするメ
 ッセージがされます。デフォルトでは、とは2つのカテゴリにかれています。たとえば、さなス
 リプト`example.py`と`python example.py -h`をしたときがあります。

```

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

```

ユーザーをけるためにをさらになセクションにけたいもあります。たとえば、すべてのオプションを1つのグループに、すべてのオプションをのグループにめることができます。のでは、するようになすることができる`--foo_*`の`--bar_*`ようなのでを。

```

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()

```

`python example.py -h`をすると、このがされます

```

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

```

docoptとdocopt_dispatchによるな

docoptとに、[docopt_dispatch]をうと、エントリポイントモジュールの`__doc__`に`--help`をること

ができます。そこでは、としてdocをしてdispatchをびすので、そのでパーサーをできます。

そうすることで、でをするわりにはいサイクロマティックなif / elseでわる、ディスパッチされたままにして、のセットをどのようにするかだけをします。

これはdispatch.onデコレータのですをトリガすべきまたはのシーケンスをえ、そのはパラメータとしてするとともにされます。

```
"""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__)
```

オンラインでコマンドラインのをむ <https://riptutorial.com/ja/python/topic/1382/コマンドラインの>

93: コメントとドキュメント

- これはのコメントです
- `print ""これはインラインコメントです`
- `"""
これは
のコメント
"""`

は[PEP257 - Docstring Conventions](#)のガイドラインにうががあります。によっては、スタイルガイド [Googleスタイルガイド](#)などやサードパーティレンダリング [Sphinx](#)などでドキュメントストリングのをしくすることもあります。

Examples

、インライン、コメント

コメントは、コードがでないにコードをするためにされます。

Pythonはコメントをするので、そこでコードをしたり、のののエラーをこしたりすることはありません。

のコメントは、ハッシュ `#` でまり、でわります。

- コメント

```
# This is a single line comment in Python
```

- インラインコメント

```
print("Hello World") # This line prints "Hello World"
```

- のにまたがるコメントは、どちらかのに`"""`または`'''`をちます。これはのとじですが、コメントとしてできます。

```
"""  
This type of comment spans multiple lines.  
These are mostly used for documentation of functions, classes and modules.  
"""
```

プログラミングによる**docstring**へのアクセス

ドキュメントストリングは、のコメントとなり、ドキュメントのとしてされているため、プログラマチックにアクセスできます。

の

```
def func():  
    """This is a function that does nothing at all"""  
    return
```

docstringには、`__doc__`をしてアクセスできます。

```
print(func.__doc__)
```

これはもしないです

```
help(func)
```

モジュール `__main__` func にするヘルプ

```
func()
```

これはもしないです

の

`function.__doc__` としての docstring であり、`help` は docstring をむにするなをします。もっとにつがあります

```
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)
```

のヘルプモジュール `__main__` greet

```
greet(name, greeting='Hello')
```

ユーザー `name` グリーティングをする

オプションのパラメータ `greeting` は、`greeting` をすることができます。

なコメントよりも **docstrings** のメリット

に docstring やのコメントをれないだけで、それほどにたなくなります。

```
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__)
```

し

```
help(greet)
```

のヘルプモジュールメインで

```
greet(name, greeting='Hello')
```

docstring をってドキュメントをく

docstringは、モジュール、クラス、メソッドをするためにされる**コメント**です。これは、それがするコンポーネントのステートメントでなければなりません。

```
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.
    """
```

docstringのは、**プログラムでアクセス**することができます。たとえば、`help`コマンドで `help`。

PEP 257

PEP 257は、**docstring**コメントのをしています。に2つのタイプがあります

- One-line Docstrings

PEP 257によれば、それらはくてなとにされるべきです。すべてが1にされます。

```
def hello():
    """Say hello to your friends."""
    print("Hello my friends!")
```

はピリオドでわらなければならない、はのでなければならない。

- のドキュメントストリング

よりい、よりな、モジュールまたはクラスには、のドキュメントストリングをするがあります。

```
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)
```

それらは、とじにあるか、のにあることができるい1のドキュメントストリングのにからまり、の
およびリストパラメーターとりをえます。

Note PEP 257は、ドキュメントストリングでどのをえなければならぬかをするものであり、ど
のフォーマットでえられるべきかをしていない。これは、のやツールがのためののをするであり
、そのいくつかはおよびこのにされています。

スフィックス

[Sphinx](#)は、ドキュメントストリングについてPythonプロジェクトのHTMLベースのドキュメント
をするツールです。されるマークアップは[reStructuredText](#)です。らはドキュメンテーションのた
めののをしています。pythonhosted.org [はそれらのにいをしています](#)。 Sphinxは、たとえば
[pyCharm IDE](#)でされます。

は、Sphinx / reStructuredTextをしてのようにされます。

```
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
```

Google Python スタイルガイド

Googleは、Pythonのコーディングをする[Google Python Style Guide](#)をしました。 Sphinx / reST
として、くのが、Googleのガイドラインにったドキュメントはがめるようにれているといいます。

の[pythonhosted.org](#)ページには、Googleスタイルガイドにってなドキュメントのがいくつかされ
ています。

Sphinxは、[Napoleon](#)プラグインをして、Googleスタイルガイドにしたでドキュメントをすることもできます。

は、Google Style Guideフォーマットをしてのようになれます。

```
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
```

オンラインでコメントとドキュメントをむ <https://riptutorial.com/ja/python/topic/4144/コメントとドキュメント>

94: コレクションモジュール

き

ビルトイン `collections` パッケージには、であり、 `dict`、 `list`、 `tuple`、 および `set` のなコレクションタイプにわるものをする、いくつかのなコレクションタイプが `set` ます。また、このモジュールは、さまざまなタイプのコレクション `MutableSet` や `ItemsView` をするクラスもします。

コレクションモジュールには、の3つのタイプがあります。

1. `UserDict`
2. ユーザーリスト
3. `UserString`

それらはそれぞれ、ばれたオブジェクトのりのラッパーとしてします。たとえば、 `UserDict` は `dict` オブジェクトのりのラッパーとしてします。いずれのも、クラスはそのきをシミュレートします。インスタンスのは、ラッパー・インスタンスのデータをしてアクセスなオブジェクトにされます。これらの3つのケースのそれぞれにおいて、これらのタイプのは、タイプからサブクラスするによってにきえられました。ただし、ラッパークラスは、になるがとしてアクセスであるため、がです。

Examples

コレクション。カウンター

`Counter` は `dict` サブクラスで、オブジェクトをにえることができます。それはあなたがえているオブジェクトのですするためのユーティリティメソッドをとっています。

```
import collections
counts = collections.Counter([1,2,3])
```

のコードは、コンストラクタにされるすべてののを `counts` というオブジェクトをします。このでは、 `Counter({1: 1, 2: 1, 3: 1})` `Counter({1: 1, 2: 1, 3: 1})`

コンストラクタの

レターカウンター

```
>>> 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})
```

ワードカウンター

```
>>> 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})
```

レシピ

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

々のをする

```
>>> c['a']
4
```

々のをする

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

カウンタのをする $4 + 2 + 0 - 3$

```
>>> sum(c.itervalues()) # negative numbers are counted!
3
```

をするのカウンタをつものだけがされる

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

0またはのをつキーをする

```
>>> c - collections.Counter()
Counter({'a': 4, 'b': 2})
```

すべてをする

```
>>> c.clear()
>>> c
Counter()
```

々のをする

```
>>> 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`は、しているキーのデフォルトをつ`dict`サブクラスをします。は、なしでびされたときにデフォルトをすでなければなりません。もされなければ、デフォルトは`None`ます。

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

`default_factory`メソッドでオブジェクトをする`defaultdict`へのをします。

`defaultdict`の**ない**は、なしでびされたときにのをすため、`str`、`int`、`list`または`dict`などのみのいずれかを`default_factory`としてすることです。

```
>>> str()
''
>>> int()
0
>>> list
[]
```

しないキーで`defaultdict`をびすと、のと同じようにエラーはしません。

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

`int`の

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2 # No errors should occur
>>> fruit_counts
defaultdict(int, {'apple': 2})
>>> fruit_counts['banana'] # No errors should occur
0
>>> fruit_counts # A new key is created
defaultdict(int, {'apple': 2, 'banana': 0})
```

のメソッドは、デフォルトの**です**

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

`default_factory`として`list`を**すると**、しいキーごとにリストが**されます**。

```
>>> 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']})
```

コレクション **.OrderedDict**

Pythonのキーのはです。するにされません。

えは

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(d)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(d)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
...

```

でされているのけは、のコードでここにしたものとなるをるかもしれないことをします。

キーがれるは、 `for` ループをするなど、されるです。

`collections.OrderedDict` クラスは、キーのをするオブジェクトをします。 `OrderedDict` は、のアイテムここでは、タプルのキーとのペアのリストとにのようにできます。

```
>>> 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)])

```

または、の `OrderedDict` をしてをすることもできます。

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])

```

`OrderedDict` することで、されたでキーアクセスがになります。

のキーにしいをりてるとどうなりますか

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

キーは、`OrderedDict`のをします。

`collections.namedtuple`

のように`namedtuple`を使っていい`Person`をします

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

2のは、タプルがつのリストです。これらのは、スペースまたはコンマでられたとしてもできます。

```
Person = namedtuple('Person', 'age, height, name')
```

または

```
Person = namedtuple('Person', 'age height name')
```

されると、なパラメータでオブジェクトをびすことによって、きタプルをインスタンスすることができます。

```
dave = Person(30, 178, 'Dave')
```

きもできます。

```
jack = Person(age=30, height=178, name='Jack S.')
```

これできタブルのにアクセスできます。

```
print(jack.age) # 30
print(jack.name) # 'Jack S.'
```

`namedtuple`コンストラクタこのでは、`'Person'` ののは `typename` です。コンストラクタとにじをするのがですが、それらはなるがあります。

```
Human = namedtuple('Person', 'age, height, name')
dave = Human(30, 178, 'Dave')
print(dave) # yields: Person(age=30, height=178, name='Dave')
```

`collections.deque`

`iterable`のデータをして、から`append`をでされたしい`deque`オブジェクトをします。 `iterable`がされていない、しい`deque`はです。

Dequeはスタックとキューの双方向に "deck" とされ、 "double-ended queue" のです。 Dequeは、どちらのでもほぼO(1)のパフォーマンスで、deque どちらからスレッドセーフ、メモリのいとおとポップをサポートします。

リストオブジェクトはのをサポートしていますが、のにされており、になるデータのサイズとのをするpop()およびinsert()、vのO(1)。

バージョン2.4の

maxlenがされていない、またはNone、 dequeはのさにするがあります。その、 dequeはされたにされます。 dequeがになると、しいがされると、するのがのからされます。 dequeは、 Unixのtailフィルタにたをします。のアクティビティーのみがのあるトランザクションやそののデータ・プールのトラッキングにもちます。

バージョン2.6でmaxlenパラメータを。

```
>>> 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'])
```

ソース <https://docs.python.org/2/library/collections.html>

コレクション.ChainMap

ChainMapはバージョン3.3でくされました

いくつかのmapsつしいChainMapオブジェクトをします。このオブジェクトは、のdictsまたはのマッピングをまとめてグループし、なのビューをします。

ChainMapはネストされたコンテキストやオーバーレイをするのにです。Pythonののは、DjangoのテンプレートエンジンのContextクラスのにあります。がのとしてわれるように、のマッピングをくリンクするのにです。しいをし、のupdate()びしをするよりもはるかにです。

ルックアップのチェーンがあるときはいつでも、ChainMapがChainMapます。には、ユーザーがしたとデフォルトのディクショナリのをつことがまれます。もう1つのは、WebでされるPOSTおよびGETパラメータマップですDjangoまたはFlask。ChainMapすると、2つのなるのされたビューがされます。

mapsパラメータリストは、にされたものからにされたものまでにべられます。ルックアップは、キーが見つかるまで、となるマッピングをします。に、きみ、およびは、のマッピングでのみします。

```
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)
```

のルックアップでがにつかるのにしてください

```
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
```

オンラインでコレクションモジュールをむ <https://riptutorial.com/ja/python/topic/498/コレクションモジュール>

95: コンテキストマネージャ "with" ステートメント

き

Pythonのコンテキストマネージャはくされていますが、そののにあるをするはほとんどいません。ファイルのみきによくされるこれらのステートメントは、システムメモリをし、のリソースがのプロセスでのみされていることをすることによってリソースをさせます。このトピックでは、Pythonのコンテキストマネージャのいをしています。

- "context_manager" "エイリアス"として、 "context_manager" "エイリアス"として*

コンテキストマネージャは、 [PEP 343](#)でされています。それらは、リソースのためのよりなメカニズムとして、 `try ... finally`する`try ... finally`よりもされることがされています。なはのりです。

このPEPでは、コンテキストマネージャは、 `__enter__()` および `__exit__()` メソッドをしています。 `__enter__()` メソッドは、 `with`のへの `__exit__()` にびされます。

に、 `with`ステートメントをのようにします。

```
with EXPR as VAR:
    BLOCK
```

ののは

```
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

コンテキストマネージャとwithステートメント

コンテキストマネージャは、コンテキストコードブロックの頭にされるオブジェクトです。、withステートメントで1つをします。それはのをします。

たとえば、ファイルオブジェクトはコンテキストマネージャです。コンテキストがすると、ファイルオブジェクトはにじられます。

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

のは、、asキーワードをしてされていas。

```
with open(filename) as open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

ブロックのがすると、コンテキストマネージャのexitメソッドがびされます。これにはがまれており、エラーによりいてるファイルやがでするにちます。ファイルやをにじないでスクリプトをすると、データがわれるなどのがするがあります。コンテキストマネージャをすることにより、このようなでやをぐためのがにじられるようにすることができます。これはPython 2.5でされました。

ターゲットにりてる

くのコンテキストマネージャは、にオブジェクトをします。そのオブジェクトをwithステートメントのしいにりてることができます。

たとえば、withステートメントでデータベースをすると、カーソルオブジェクトがられます。

```
with database_connection as cursor:
    cursor.execute(sql_query)
```

ファイルオブジェクトはそれをします。これにより、ファイルオブジェクトをいて、1つのでコンテキストマネージャとしてすることができます。

```
with open(filename) as open_file:
    file_contents = open_file.read()
```

のコンテキストマネージャをく

コンテキストマネージャは、__enter__()と__exit__() 2つのマジックメソッドをするオブジェクトですのメソッドもできます。

```
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
```

コンテキストがでた、そのにするはtriple `exc_type`、`exc_value`、`traceback` これらは `sys.exc_info()` によってされるものとじですとして `sys.exc_info()` れます。コンテキストがにする、これらのの3つはすべて `None` ます。

がし、`__exit__` メソッドにされた、このメソッドはをするために `True` をすことができます。そうしないと、は `__exit__` のにします。

```
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
```

2のでは、`with-statement` のでがしても、がスコープにするに `__exit__` ハンドラがまだされていることにしてください。

`__exit__` メソッドだけがなは、コンテキストマネージャのインスタンスをすことができます

```
class MyContextManager:
    def __enter__(self):
        return self

    def __exit__(self):
        print('something')
```

ジェネレータをしてのコンテキストマネージャをする

`contextlib.contextmanager` デコレータのおかげで、ジェネレータをとってコンテキストマネージャをくこともできます

```
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))

```

をする

```

Enter
Right in the middle with cm = 3
Exit

```

デコレータは、ジェネレータを1にすることによってコンテキストマネージャをするをします。
`yield`ののすべてが`__enter__`メソッドになり、`__enter__`されたはジェネレータによってされたになります
`with`ステートメントのにバインドできます `__exit__`ののすべてが`__exit__`メソッドになります。

は、コンテキストマネージャによってするがある、`try..except..finally` -blockは、でくことができ、にしたすべての`with` -ブロックは、このブロックによってされます。

```

@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)

```

これにより、

```

Enter
Dividing by cm = 0
Caught error
Cleaning up
Exit

```

のコンテキストマネージャ

のコンテンツマネージャをにくことができます。

```

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')
```

コンテキストマネージャのネストとじがあります。

```
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')
```

リソースの

```
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__()` メソッドはオブジェクトをします。この、ファイルとモードをオープンファイルにします。 `__enter__()` がき、ファイルとす `__exit__()` だけで、それをクローズします。

これらのマジックメソッド `__enter__`、 `__exit__` をう `with`、 `with`ステートメントでにできるオブジェクトをできます。

ファイルクラスをする

```
for _ in range(10000):
    with File('foo.txt', 'w') as f:
        f.write('foo')
```

オンラインでコンテキストマネージャ "with"ステートメントをむ

<https://riptutorial.com/ja/python/topic/928/コンテキストマネージャ---with-ステートメント->

96: サブプロセスライブラリ

- `subprocess.callargs`、*、`stdin = None`、`stdout = None`、`stderr = None`、`shell = False`、`timeout = None`
- サブプロセス.`Popenargs`、`bufsize = -1`、`=なし`、`stdin =なし`、`stdout =なし`、`stderr =なし`、`preexec_fn =なし`、`close_fds = True`、`shell = False`、`cwd =なし`、`env =なし`、`universal_newlines = False`、`startupinfo = None`、`creationflags = 0`、`restore_signals = True`、`start_new_session = False`、`pass_fds =`

パラメーター

パラメータ	
<code>args</code>	のファイル、またはファイルとのシーケンス - 'ls'、 ['ls', '-la']
<code>shell</code>	シェルのでしますか POSIXの/bin/shへのデフォルトシェル。
<code>cwd</code>	プロセスのディレクトリ。

Examples

コマンドのびし

もなは、`subprocess.call`のです。のとしてリストをくれます。リストののは、びすアプリケーションでなければなりません。リストののは、そのアプリケーションにされるです。

```
subprocess.call([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

シェルコマンドののは、`shell=True`とし、コマンドをリストではなくとしてします。

```
subprocess.call('echo "Hello, world"', shell=True)
```

の2つのコマンドは、サブプロセスの`exit status`のみをします。さらに、`shell=True`をするときはセキュリティがあるのでしてください [ここを](#)。

あなたがサブプロセスのをすることができるようになりたいは、わり`subprocess.call`し`subprocess.check_output`。よりないについては、[これをして](#)ください。

Popenによるの

`subprocess.Popen`をすると、されたプロセスを`subprocess.call`よりかくできます。

サブプロセスの

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

`Popen`のシグネチャは、`call`に似ていて`call`。しかし、`Popen`は`call`のようにサブプロセスがするのを終わりに、すぐに帰ります。

するためにサブプロセスをっている

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
process.wait()
```

サブプロセスからののみみ

```
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
```

のサブプロセスへのアクセス

サブプロセスがしていなくても、`stdin`と`stdout`をみきすることができます。これは、のプログラムのをするにです。

サブプロセスへのきみ

```
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.
```

ただし、なやりとりではなく、のセットが1つだけなのは、`stdin`と`stdout`アクセスするのではなく、`communicate()`をするがあります。

サブプロセスからストリームをみる

サブプロセスのをでするは、のスニペットをできます。

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

サブコマンドにEOLがまれていない、のスニペットはしません。されたをのようになでみるができます。

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

1のとしてされた`read`は、に1つのをむためにんでえま。のをして、なのをみるようにすることができます。のまたは0は、EOFにするまで、のとして`read`ようにし`read` [ここを](#)。

ののスニペットでは、サブプロセスがするまで`process.poll()`は`None`なります。これは、みせるがなくなったらループをするためにされます。

のをサブプロセスの`stderr`にすることができます。

コマンドリストのをする

コマンドをできるサブプロセスメソッドは、なくとも`shell_mode=True`をして、リストのコマンドをとします。

リストをするためのルールは、になコマンドのは、ずしもではありません。いにも、それをにするになツールがあります `shlex`。コマンドとしてするリストをするもなはのとおりです。

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

な

```
import shlex
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

オンラインでサブプロセスライブラリをむ <https://riptutorial.com/ja/python/topic/1393/サブプロセスライブラリ>

97: ジャンゴ

き

Djangoは、なとクリーンでなをする、のPython Webフレームワークです。かなによってされているため、Webのをにしているため、アプリケーションをきすはありません。それはでオープンソースです。

Examples

DjangoのHello World

あなたのdjangoをつてなHello Worldをりましょう。

にあなたのPCにdjangoがインストールされていることをしましょう。

ターミナルをき、のようします。python -c "import django"
->エラーがないは、djangoがにインストールされていることをします。

すぐdjangoでプロジェクトをしましょう。ターミナルでのコマンドをくには
django-admin startproject HelloWorld

のコマンドは、HelloWorldというのディレクトリをします。

ディレクトリはのようになります

こんにちは

```
| --helloworld  
|| - init .py  
|| --settings.py  
|| --urls.py  
|| --wsgi.py  
| --manage.py
```

ライティングビュー djangoのドキュメントからの

ビューまたはビューは、にWebリクエストをけり、WebレスポンスをすPythonです。このレスポンスは、WebページなどのHTMLコンテンツにすることができます。ドキュメンテーションには、プロジェクトディレクトリにされたviews.pyでしたほうがよいがあります。

こんにちはのメッセージをすビューですviews.py

```
from django.http import HttpResponse  
  
def helloWorld(request):  
    return HttpResponse("Hello World!! Django Welcomes You.")
```

コードをにしましょう。

- まず、django.httpモジュールからHttpResponseクラスをインポートします。
- に、helloWorldというをします。これはビューです。ビューは、のパラメータとしてHttpRequestオブジェクトをります。、requestというがけられています。

ビューのはありません。 Djangoがそれをするためには、あるでをけるはありません。 たちはそれをhelloWorldとんでいたなので、それがをするのかははっきりしています。

- このビューは、されたをむHttpResponseオブジェクトをします。ビューは、HttpResponseオブジェクトをすがあります。

[djangoビューのについては、ここをクリックしてください](#)

URL をビューにマッピングする

このビューをのURLにするには、URLconfをするがあります。

そのに、djangoがどのようにリクエストをするのかをしましょう。

- DjangoはするルートURLconfモジュールをします。
- DjangoはそのPythonモジュールをロードし、urlpatternsをします。これはPythonのdjango.conf.urls.urlインスタンスのリストでなければなりません。
- DjangoはURLパターンをにし、されたURLにするのパターンでします。
- の1つがすると、Djangoはえられたビューをインポートしてびします。これはなPythonです。

URLconfのはのとおりで。

```
from django.conf.urls import url
from . import views #import the views.py from current directory

urlpatterns = [
    url(r'^helloworld/$', views.helloWorld),
]
```

[django Urlsのについては、ここをクリックしてください](#)

HelloWorldにディレクトリをして、のコマンドをにきします。

python manage.py runserver

デフォルトでは、サーバーは127.0.0.1:8000でされます

ブラウザをき、127.0.0.1:8000/helloworld/とします。このページには "Hello World !! Django Welcome You"とされます。

[オンラインでジャンゴをむ](https://riptutorial.com/ja/python/topic/8994/ジャンゴ) <https://riptutorial.com/ja/python/topic/8994/ジャンゴ>

98: スタック

き

スタックは、ラスト・イン・ファーストアウトLIFOによっておよびされるオブジェクトのコンテナです。プッシュダウンスタックでは、アイテムをスタックにプッシュし、スタックからアイテムをポップするという2つのだけがされます。スタックは、されたアクセスデータです。は、スタックののみまたはできます。スタックのはのとおります。スタックはであるか、スタックでされ、スタックはスタックでされます。

- `stack = []`スタックをする
- `stack.append(object)`スタックのにオブジェクトをする
- `stack.pop()` -> `object`スタックからのオブジェクトをし、それもします
- `list[-1]` -> `object`のオブジェクトをせずにそのオブジェクトをいみます

ウィキペディアから

コンピュータサイエンスでは、スタックは、をコレクションにする`push`と、まだされていないもされたをする`pop`の2つのなをつ、のコレクションとしてするデータです。

それらのがアクセスされるにより、スタックはラスト・イン・ファーストアウト *LIFO* スタックともされます。

Pythonでは、リストをスタックとしてし、`append()`を`push`および`pop()`として`pop`としてできます。のはO(1)でされます。

Pythonの`deque`データは、スタックとしてもできます。デキューはリストとして、`deque`からののさでプッシュとポップをにします。

Examples

ListオブジェクトをしたStackクラスの

`list`オブジェクトをすると、スタックがであるかどうかをべたりしたりするヘルパーメソッドを使って、にするジェネリックスタックをできます。のためのPythonドキュメントチェック `list`としてStack [こちらを](#)。

```
#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

```

```

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())

```

```

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

```

の

くの、スタックはにされます。なは、カッコのがしているかどうかをすることです。

たとえば、`([])`は、とのがペアをするため、しています。`()<>`はしません。にはパートナーがないためです)。`[]]`もマッチしません。なぜなら、ペアはにのペアのかでなければならないからです。

```
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()
```

オンラインでスタックをむ <https://riptutorial.com/ja/python/topic/3807/スタック>

99: セキュリティと

き

Pythonは、コンピュータセキュリティとネットワークセキュリティでもなの1つで、セキュリティとにきなをめています。このトピックでは、コンピュータとネットワークのセキュリティ、ハッシング、/アルゴリズムにされているPythonのとについてします。

- `hashlib.newname`
- `hashlib.pbkdf2_hmac`、パスワード、ソルト、ラウンド、`dklen` = なし

`hashlib`くの方法では、ではなくバイトのバッファとしてなをすがあります。これは、`hashlib.new().update()` および `hashlib.pbkdf2_hmac` です。があるは、`b`をのにすることで、をバイトバッファにすることができます。

```
"This is a string"
b"This is a buffer of bytes"
```

Examples

メッセージダイジェストの

`hashlib`モジュールでは、`new`方法をしてメッセージダイジェストジェネレータをできます。これらのジェネレータはのをダイジェストにします

```
import hashlib

h = hashlib.new('sha256')
h.update(b'Nobody expects the Spanish Inquisition.')
h.digest()
# ==>
b'\x0f\xda\xdaVR[\x12\x90\xff\x16\xfb\x17D\xcf\xb4\x82\xdd)\x14\xff\xbc\xb6Iy\x0c\x0eX\x9eF-'
=
```

きなファイルチャンクをチャンクでハッシュするのにな `digest` をびすに、`update` をのびすことができます。 `hexdigest` をってダイジェストを16ですすることもできます

```
h.hexdigest()
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbc649790c0e589e462d3d'
```

なハッシングアルゴリズム

`hashlib.new`は、ジェネレータをするためにアルゴリズムをびすときにアルゴリズムのがです。のPythonインタプリタでなアルゴリズムをべるには、`hashlib.algorithms_available`して

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'}
```

されるリストは、プラットフォームとインタプリタによってなります。アルゴリズムがであることをしてください。

hashlib.algorithms_guaranteed をしてな、すべてのプラットフォームとインタプリタでできることがされているアルゴリズムもあります。

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

なパスワードのハッシュ

hashlib モジュールによってされた [PBKDF2 アルゴリズム](#) をして、なパスワードハッシングをできます。このアルゴリズムでは、されたハッシュからのパスワードをするためにをぐことはできませんが、そのようなはにになります。

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 はのダイジェストアルゴリズムでできますが、のではされる SHA256 をしています。ランダムなハッシュされたパスワードとにされるべきです。されたパスワードとされたハッシュをするために、もうそれをとします。パスワードはなるでハッシュされることがです。ラウンドについては、[アプリケーションのなりく](#) することをおめします。

を16でするには、binascii モジュールをします。

```
import binascii
hexhash = binascii.hexlify(hash)
```

PBKDF2 はくないが、[bcrypt](#)、に [scrypt](#) はbrute-force にしてよりであるとえられている。のところ、どちらもPython ライブラリではありません。

ファイルハッシング

ハッシュは、シーケンスのバイトをシーケンスにします。ファイルをハッシュすることは、くのであります。ハッシュをして、2つのファイルがであるかどうかをチェックしたり、ファイルのがまたはされていないことをしたりすることができます。

hashlibを使ってファイルのハッシュをすることができます

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

大きなファイルの、バッファをします。

```
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())
```

pycryptoをした

Pythonにみまれているは、ハッシングにされています。には[pycrypto](#)のようなモジュールがです。えは、これは、のためのとえられる[AESアルゴリズム](#)をする。のコードは、パスフレーズをしてのメッセージをします。

```
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)
```

AESアルゴリズムは、キー、ベクトルIV、およびされるのメッセージという3つのパラメータをとります。あなたがランダムにされたAESキーを持っているならば、それをして、ランダムベクトルをするだけです。しかしパスフレーズはしいサイズをとっておらず、にランダムではなく、したがってエントロピーはさいとすれば、することはできません。わりに、 [PBKDF2アルゴリズムのみ](#)

みをして、パスワードから128ビットのベクトルと256ビットのキーをします。

されたメッセージごとになるベクトルとをつことがランダムなにしてください。これにより、に2つのしいメッセージがのされたテキストにならないことがされますが、はのパスフレーズでされたメッセージにしてあるパスフレーズをしたをできません。のためにじベクトルとをするために、このはされたメッセージとにされなければならない。

のコードは、たちのメッセージをします

```
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:])
```

pycryptoをしたRSAの

RSAをしてメッセージをできます。なは、のRSAへのアクセスでのみすることができ、では、するだけである。があなたのをっているり、あなたはされたメッセージをでき、されません。たとえば、メールにされるです。、このにはpycryptoなどのサードパーティのモジュールがです。

```
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)
```

のはですが、ではなくをします。

```
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')
```

のでは、になPKCS1 v1.5アルゴリズムをしています。 pycryptoはしいPKCS1 PSSアルゴリズムもしています PKCS1_v1_5をPKCS1_PSSできえと、そのをしたいにします。、 [それをするはほとんどない](#)ようです。

pycryptoをしたRSA

には、メッセージのとをすることなく、メッセージをできるというがあります。はのをるだけでよく、されたするをっているだけがそれをできるようにメッセージをすることができます。、このには[pycrypto](#)などの[サードパーティ](#)のモジュールがです。

```
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)
```

は、しいをっているは、メッセージをできます。

```
with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
decrypted = cipher.decrypt(encrypted)
```

のでは、PKCS1 OAEPをしています。 pycryptoはPKCS1 v1.5もしていますが、これは[の](#)のためしいプロトコルにはされません。

[オンラインでセキュリティとをむ](#) <https://riptutorial.com/ja/python/topic/2598/セキュリティと>

100: セット

- `empty_set = set`のセットをする
- `literal_set = {'foo', 'bar', 'baz'}`に3つのをむセットをする
- `set_from_list = set(['foo', 'bar', 'baz'])`しいセットにしてsetをびす
- `set_from_iter = set(30のxはxのiterablesをしてをする)`
- `set_from_iter = {xは10のiにして[random.randint(0,10)]のxにして}}`

セットはであり、になルックアップをっていますテクニカルをするは、 $O(1)$ 。あなたがのコレクションをっているときに、はありません、あなたはくのでをしているでしようすることはらしいです。インデックスでをするがなは、わりにリストをすることをしてください。がなは、リストもしてください。

セットはであり、したがってハッシュすることはできません。したがって、それらをディクショナリ・キーとしてすることも、のセットにれることも、ハッシュタイプをとするそののにくこともできません。そのような、の `frozenset` することができます。

セットのはハッシュでなければなりません。これは、しい `__hash__` メソッドをっていることをします。これは `__hash__` としてい `__eq__` 。に、`list` や `set` などのハッシュではなく、セットにれることはできません。このがしたは、`dict` とキーのをしてください。

Examples

リストののをする

あなたがレストランのリストをっているとしましょう - あなたはファイルからそれをんでいます。あなたはリストのユニークなレストランをにします。のをリストからするもいは、それをセットにすることです。

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
unique_restaurants = set(restaurants)
print(unique_restaurants)
# prints {'Chicken Chicken', 'McDonald's', 'Burger King'}
```

セットはのリストとじではないことにしてください。セットがけられていないので、それはじように、ある `dict`。

これはPythonのみ `list` をって `List` ににすることができます。これは、のリストとじリストですがはありません

```
list(unique_restaurants)
# ['Chicken Chicken', 'McDonald's', 'Burger King']
```

これを1にするのもです。

```
# Removes all duplicates and returns another list
list(set(restaurants))
```

これで、のリストでできるはすべてできます。

セットにする

のセットと

```
# 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
```

で

```
# 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!
```

セットではしいセットがされますが、するインプレースバージョンがあります。

	インプレース	インプレース
	$s \mid= t$	
	$s = t$	intersection_update
	$s -= t$	difference_update
symmetric_difference	$s \wedge t$	symmetric_difference_update

例えば

```
s = {1, 2}
s.update({3, 4})    # s == {1, 2, 3, 4}
```

セットとマルチセット

はなるのです。しかし、には、ずしもではなく、のをしているのけられていないをうことができます。

このをえてみましょう。

```
>>> setA = {'a','b','b','c'}
>>> setA
set(['a', 'c', 'b'])
```

'a'、'b'、'b'、'c'をセットデータにすることにより、'b'が2するというをってしまった。もちろん、をリストにすると、このがされます

```
>>> listA = ['a','b','b','c']
>>> listA
['a', 'b', 'b', 'c']
```

リストのデータはななけをし、がくなります。

マルチセットをするために、Pythonは、collectionsモジュールからCounterクラスをしますバージョン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は、がキーとしてされ、そのカウントがとしてされるです。そして、すべてのとして、それはべえられていないコレクションです。

メソッドとみみをしてをする

2の a と b をする

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

`{1}`は1つののセットをしますが、`{}`はの`dict`します。のセットをするしいは`set()`です。

`a.intersection(b)`、のにするでしいセットをし a と b

```
>>> a.intersection(b)
{3, 4}
```

`a.union(b)`のいずれかににするでしいセットをし a と b

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

`a.difference(b)`は、が b に a が b はしないしいをす

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

`a.symmetric_difference(b)`は、 a または b いずれかににするがにはしないをつしいセットをします

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

```
a.symmetric_difference(b) == b.symmetric_difference(a)
```

サブセットとスーパーセット

`c.issubset(a)`は、 c が a かどうかをテスト a ます。

`a.issuperset(c)`は、 c が a かどうかをテスト a ます。

```
>>> c = {1, 2}
>>> c.issubset(a)
```

```
True
>>> a.issuperset(c)
True
```

のには、にすようにのがあります。

	オペレーター
<code>a.intersection(b)</code>	<code>a & b</code>
<code>a.union(b)</code>	<code> b</code>
<code>a.difference(b)</code>	<code>a - b</code>
<code>a.symmetric_difference(b)</code>	<code>a ^ b</code>
<code>a.issubset(b)</code>	<code>a <= b</code>
<code>a.issuperset(b)</code>	<code>a >= b</code>

な

a と d は、 a の d もなく、そのの d もないである。

```
>>> 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
```

メンバーシップのテスト

みみ`in` occurrencesためのキーワード

```
>>> 1 in a
True
>>> 6 in a
False
```

さ

みみの `len()` は、セットののをします

```
>>> len(a)
4
>>> len(b)
3
```

セットのセット

```
{{1,2}, {3,4}}
```

につながる

```
TypeError: unhashable type: 'set'
```

わりに、 `frozenset` してください

```
{frozenset({1, 2}), frozenset({3, 4})}
```

オンラインでセットをむ <https://riptutorial.com/ja/python/topic/497/セット>

101: ソケット

き

くのプログラミングは、ソケットをしてプロセスまたはデバイスでします。このトピックでは、Pythonのソケットモジュールをして、なネットワークプロトコルをとってデータをできるようにするについてします。

パラメーター

パラメータ	
socket.AF_UNIX	UNIXソケット
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

Examples

UDPのデータ

UDPはコネクションレスプロトコルです。のプロセスまたはコンピュータへのメッセージは、らかのをすることなくされます。あなたのメッセージがされた、はありません。UDPは、レイテンシになアプリケーションや、ネットワークのブロードキャストをするアプリケーションでされます。

のコードは、UDPをしてローカルホストのポート6667でしているプロセスにメッセージをします

UDPはコネクションレスなので、にソケットを「じる」はないことにしてください。

```
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))
```

UDPによるデータ

UDPはコネクションレスプロトコルです。つまり、メッセージをするピアは、メッセージをする

にをするはありません。 `socket.recvfrom` はタプルをす `msg` [ソケットがけたメッセージ]、 `addr` [のアドレス]

`socket` モジュールのみをするUDPサーバ

```
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))
```

は、 `socketserver.UDPServer` をしたの `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()
```

デフォルトでは、 `sockets` ブロックされます。つまり、スクリプトのは、ソケットがデータをけるまでします。

TCPのデータ

インターネットをしてデータをすることは、のモジュールをしてになります。ソケットモジュールは、のコンピュータやプロセスからのデータのをう、となるオペレーティングシステムオペレーションへのレベルのアクセスをします。

のコードは、ホスト `localhost` のポート `6667` でしているTCPサーバにバイト `b'Hello'` し、にをします。

```
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()
```

ソケットはデフォルトでブロックされています。つまり、プログラムはがするまでし、アクションがするまでびしをします。の、サーバがにをけることをします。の、オペレーティングシステムにデータをキューイングしてでするのになバッファスペースがあることをします。

ソケットははにじてください。

マルチスレッドTCPソケットサーバー

なしですと、このプログラムはポート5000 127.0.0.1へのをリスンするTCPソケットサーバーをします。サーバーは、のスレッドでをします。

-cきをしてすると、このプログラムはサーバーにし、クライアント・リストをみり、それをします。クライアントリストはJSONとしてされます。クライアントは、-nをすことでできます。なるをすことにより、クライアントリストへのがされることがあります。

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()

```

サーバー

```

$ python client_list.py
Starting server...

```

クライアント

```

$ python client_list.py -c -n name1
{
    "name1": {
        "address": "127.0.0.1",
        "port": 62210,
        "name": "name1"
    }
}

```

バッファは1024バイトにされています。クライアントリストのJSONがこのサイズをえと、りてられます。これにより、のがします。

```

ValueError: Unterminated string starting at: line 1 column 1023 (char 1022)

```

LinuxのRawソケット

まず、ネットワークカードのチェックサムをにします。

```

sudo ethtool -K eth1 tx off

```

に、SOCK_RAWソケットをしてパケットをします。

```

#!/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)
```

オンラインでソケットをむ <https://riptutorial.com/ja/python/topic/1530/ソケット>

102: そののからステートメントをりえるための

のとしてPythonでのNOスイッチはありません。されたをカバーするPEP [PEP-3103](#) がありました。

Pythonでのswitchをするにするレシピのリストがたくさんあります。ここでは、もなオプションをしようとしています。するはのとおりで。

- <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

がするものをしてくださいif / else。

さて、switch / caseがなは、いold if / elseをするのがもな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!")
```

それはにえるかもしれませんが、いつもかわいいとはりませんが、これははるかになです。

```
>>> 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
```

のをう

もうつのなは、のをすることです。

```
switch = {
    1: lambda: 'one',
    2: lambda: 'two',
```

```
42: lambda: 'the answer of life the universe and everything',
}
```

デフォルトをします

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

のgetメソッドをして、をえられたをしてチェックしてします。valueがにしない、default_caseがされます。

```
>>> 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
```

あなたはなをとってスイッチがよりくえるようにすることもできます

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

>>> run_switch(1)
one
```

クラスのイントロスペクションをする

クラスをして、スイッチ/ケースをすることができます。は、をインスタンスのバインドされたメソッドにするgetattr()をしてクラスのイントロスペクションをして、"case"をします。

そのイントロスペクティングメソッドは、__call__メソッドにエイリアスされて、()がオーバーロードされます。

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

    __call__ = switch
```

に、それをえくするために、SwitchBaseクラスをサブクラスしSwitchBaseただし、1つのクラスでうことができます。すべてのcaseをメソッドとしてします。

```
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!')

```

それで私たちはにそれをうことができます

```

>>> 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!

```

コンテキストマネージャをする

にみやすくですが、if/elseよりはるかにでないもう1つのは、のようなクラスをし、それをしてをし、コンテキストでびしなものとしてしますされたとするはtrueをします。

```

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

```

ケースをすることは、の`switch / case`コンストラクトののであるほうがですにほぼしています。

```

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!')

```

はのようになります。


```
>>> 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
```

ノートベネ

- このソリューションは、[pypi](#)でなスイッチモジュールとしてされています。

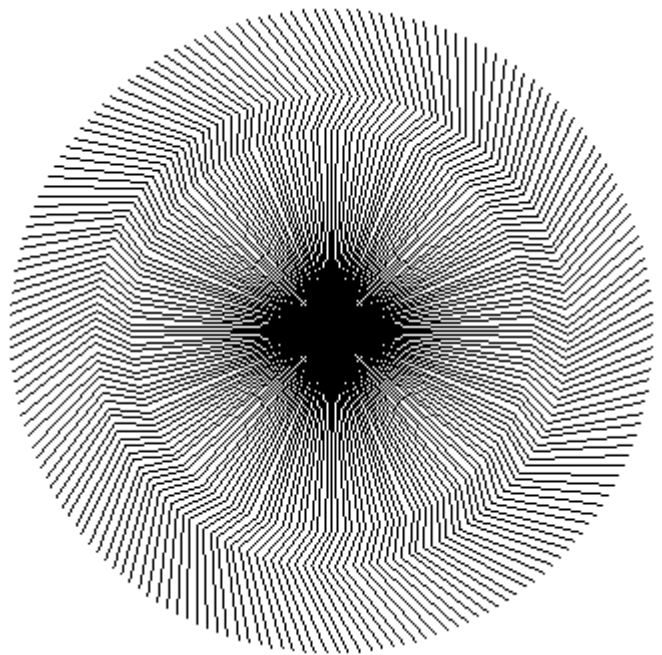
オンラインでそののからステートメントをりえるためのをむ

<https://riptutorial.com/ja/python/topic/4268/そののからステートメントをりえるための>

103: タートルグラフィックス

Examples

ツイスト タートルグラフィックス



ここでは、タートルグラフィックスのツイスト

```
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()
```

オンラインでタートルグラフィックスをむ <https://riptutorial.com/ja/python/topic/7915/タートルグラフィックス>

104: タイプヒント

- `typing.Callable` `[[int、str]、None]` - `> def func` `aint、bstr` - `> なし`
- マッピング `[str、int]` - `> {"a"1、 "b"2、 "c"3}`
- `typing.List` `[int]` - `> [1、 2、 3]`
- `typing.Set` `[int]` - `> {1,2,3}`
- `typing.Optional` `[int]` - `> なし` または `int`
- シーケンス `[int]` - `> [1,2,3]` または `1,2,3` をします。
- `typing.Any` - `> の`
- `typing.Union` `[int、 str]` - `> 1` または `"1"`
- `T = typing.TypeVar 'T'` - `>`

タイプヒントは、[PEP 484](#)でされているように、Pythonコードののをにすなソリューションです。タイプヒントは、`typing`モジュールのにされることにより、Pythonユーザにコードにをけるをし、タイプチェッカーをしなから、にコードにをします。

Examples

なタイプ

`typing.TypeVar`はジェネリックのファクトリです。なは、ジェネリック/クラス/メソッドアノテーションのパラメータ/プレースホルダとしてすることです。

```
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]
```

にをする

2つのをけり、そのをすをすのをてみましょう

```
def two_sum(a, b):
    return a + b
```

このコードをすることで、`two_sum`のをかつ`two_sum`すことはできません。`int`をすると、ともします

```
print(two_sum(2, 1)) # result: 3
```

とで

```
print(two_sum("a", "b")) # result: "ab"
```

`list`や`tuple`などのものとします。

このようなPythonのなのために、えられたにくのものがされる、どのチェッカーでもこののびしをするかどうかをにできません。

チェッカーをけるために、たちがするをすのヒントをできるようになりました。

`int`のみをすることをすために、をのようにすることができます。

```
def two_sum(a: int, b: int):  
    return a + b
```

アノテーションはのにされ、`a:`でられます。

に、`str`のみがされていることをすために、をしてをします。

```
def two_sum(a: str, b: str):  
    return a + b
```

のをするだけでなく、びしのりをすこともできます。これは、リストののじのの->のにをすることによってわれますが、のには:に

```
def two_sum(a: int, b: int) -> int:  
    return a + b
```

ここで、`two_sum`をびすときのりは`int`でなければならないことをしました。に、`str`、`float`、`list`、`set`などのなをできます。

ヒントはほとんどがチェッカーやIDEでされますが、それらをするがあることがあります。これは`__annotations__`をしてうことができます

```
two_sum.__annotations__  
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

クラスメンバーとメソッド

```
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))
```

がされたときにアノテーションがされるので、のクラスのがです。は、インポートされたにインポートをきこすクラスをするときにもできます。

と

にはをしてがけられます。

```
x = 3 # type: int
x = negate(x)
x = 'a type-checker might catch this error'
```

Python 3.x 3.6

Python 3.6から、[のしいもあります](#)。のコードはフォームをするかもしれません

```
x: int = 3
```

コメントとはなり、をせずに、にされていないにヒントをすることもできます。

```
y: int
```

さらに、これらがモジュールまたはクラスレベルでされているは、`typing.get_type_hints(class_or_module)`をしてタイプヒントをできます。

```
class Foo:
    x: int
    y: str = 'abc'

print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

あるいは、`__annotations__`またはをしてアクセスすることもできます。

```
x: int
print(__annotations__)
# {'x': <class 'int'>}

class C:
    s: str
print(C.__annotations__)
# {'s': <class 'str'>}
```

NamedTuple

ヒントきのきタプルのは、`typing`モジュールのNamedTupleをっています

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

のはののですが、チェッカーのをにするためにじのにりてるがあります。

キーワードのタイプヒント

```
def hello_world(greeting: str = 'Hello'):
    print(greeting + ' world!')
```

キーワードがどのようにスタイルされているかとはに、のりのスペースにしてください。

オンラインでタイプヒントをむ <https://riptutorial.com/ja/python/topic/1766/タイプヒント>

105: タプル

き

タプルはののリストです。タプルは、Pythonのもでもなコレクションの1つで、カンマ `value = 1, 2, 3` でできます。

- 1、a、"hello"aはでなければならぬ
- のタプル
- 1、1タプル。 `(1)` はタプルではありません。
- 1、2、33タプル `1,2,3`

カッコは、のタプルまたはびしでするにのみです。

タプルはののです。はのにすることができ、それらはでインデックスけされるので、タプルはリストのようなものです。ないは、タプルはでハッシュであるため、セットとマップでできることです

Examples

タプルのけ

```
x = (1, 2, 3)
x[0] # 1
x[1] # 2
x[2] # 3
x[3] # IndexError: tuple index out of range
```

のをつインデックスは、のから-1としてします。

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

のをけする

```
print(x[:-1]) # (1, 2)
print(x[-1:]) # (3,)
print(x[1:3]) # (2, 3)
```

タプルはです

Pythonの`list`と`tuple`のないの1つは、タプルがであることです。つまり、タプルがされたにアイテムをまたはすることはできません。え

```
>>> 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
```

に、タプルはありません。`.append`と`.extend`のようなを`list`ありません。 `+=`をうことはですが、タプルではなくのバインディングをします

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

タプルに`lists`などのオブジェクトをするときはしてください。これは、それらをするときににするにつながるがあります。え

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

ともエラーをこし、タプルのリストのをします

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

`+=`をしてタプルに「」することができます。これは、したいでしいタプルをし、それをのにりてることによってします。いタプルはされませんが、きえられます

これはリストとのをけますが、これはく、にするはいです。

タプルはにハッシュであり、しい

```
hash( (1, 2) ) # ok
hash( ([], {"hello"}) ) # not ok, since lists and sets are not hashable
```

したがって、タプルは、そののそれぞれができるにのみ、 `set`にれたり、または`dict`としてキーとしてれることができます。

```
{ (1, 2) } # ok
{ ([], {"hello"}) } # not ok
```


タプル

、タプルはコンマでられたのリストです。

```
t = 'a', 'b', 'c', 'd', 'e'
```

ではありませんが、タプルをカッコでむのがです。

```
t = ('a', 'b', 'c', 'd', 'e')
```

カッコでのタプルをする

```
t0 = ()  
type(t0)          # <type 'tuple'>
```

のをつタプルをするには、のカンマをめるがあります。

```
t1 = 'a',  
type(t1)          # <type 'tuple'>
```

カッコののはタプルではないことにしてください。

```
t2 = ('a')  
type(t2)          # <type 'str'>
```

シングルトンタプルをするには、にカンマがです。

```
t2 = ('a',)  
type(t2)          # <type 'tuple'>
```

シングルトンタプルのは、カッコをすることをおめします [のカンマのPEP8](#)を。また、のコンマのにがない [のはPEP8](#)を

```
t2 = ('a',)          # PEP8-compliant  
t2 = 'a',            # this notation is not recommended by PEP8  
t2 = ('a', )         # this notation is not recommended by PEP8
```

タプルをするのはみみ`tuple`です。

```
t = tuple('lupins')  
print(t)              # ('l', 'u', 'p', 'i', 'n', 's')  
t = tuple(range(3))  
print(t)              # (0, 1, 2)
```

これらのは、[Allen B.Dayey](#)の「[Think Python](#)」のについています。

タプルのパッキングとアンパッキング

Pythonのタプルはコンマでられたです。タプルをするためのみはオプションであるため、2つの

```
a = 1, 2, 3 # a is the tuple (1, 2, 3)
```

そして

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

です。a = 1, 2, 3は、タプルでをまとめるため、パッキングともばれます。

1のタプルもタプルであることにしてください。がのではなくタプルであることをPythonにえるために、のコンマ

```
a = 1 # a is the value 1
a = 1, # a is the tuple (1,)
```

カッコをするはカンマもです

```
a = (1,) # a is the tuple (1,)
a = (1) # a is the value 1 and not a tuple
```

タプルからをアンパックし、のをうには

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

シンボル_は、プレースホルダのをたすタブルのがなにいてのとしてできます。

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

タプル

```
x, = 1, # x is the value 1
x = 1, # x is the tuple (1,)
```

Python 3では、*をつターゲットを[catch-all](#)としてできます「[パッケージの](#)」を。

Python 3.x 3.0

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

の

タブルのをにする

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

または、をするは、をタブルにする

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

ビルト イン タブル

タブルはのみみをサポートしています

がじの、Pythonはをし、をします。がなるである、がかどうかをチェックします。

- のはをいます。
- いずれかのがのは、もうのがされます。
- そうでなければ、はアルファベットにソートされます。

リストの1つのわりにすると、いリストは「よりきい」ものになります。のリストがじは、0をします。

```
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
```

タブル

lenは、タブルのをします

```
len(tuple1)
Out: 5
```

タプルの

`max`はタプルのを`max`します

```
max(tuple1)
Out: 'e'

max(tuple2)
Out: '3'
```

タプルの

`min`は、をつタプルからアイテムをします

```
min(tuple1)
Out: 'a'

min(tuple2)
Out: '1'
```

リストをタプルにする

ビルトイン`tuple`、リストを`tuple`します。

```
list = [1,2,3,4,5]
tuple(list)
Out: (1, 2, 3, 4, 5)
```

タプル

`+`をして2つのタプルをする

```
tuple1 + tuple2
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

オンラインでタプルをむ <https://riptutorial.com/ja/python/topic/927/タプル>

106: ディスクリプタ

Examples

な

ディスクリプタには2のがあります。データは、`__get__()` `__set__()` メソッドと `__set__()` メソッドのをするオブジェクトとしてされますが、データは `__get__()` メソッドのみをします。これは、オーバーライドとインスタンスのをするにです。データとインスタンスのエントリがじをするは、データがされます。ただし、データとインスタンスのエントリがじをする、インスタンスのエントリがされます。

びされたときには `AttributeError` をげるセット のを と **set** をし、みりのデータをするには。をさせる **placeholder** をつ **set** メソッドをするだけで、データにすることができます。

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

された

```
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
```

のコンバージョン

Descriptorオブジェクトは、するオブジェクトがににすることをします。

えられたヘルツとのオシレータをモデルしたいとします。をするにはをし、をするはをする

```
>>> 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
```

、ヘルツの1つを「アンカー」、つまりなしでできるものとしてし、そのためのディスクリプタクラスをします。

```
class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)
```

「その」の、は、アンカーのからされます。コンバージョンをうクラスをします。

```
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)
```

これで、Oscillatorクラスをくことができます

```
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
```

オンラインでディスクリプタをむ <https://riptutorial.com/ja/python/topic/3405/ディスクリプタ>

107: ディスモジュール

Examples

disモジュールの

```
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', '...
```

Pythonバイトコードとはですか

Pythonはハイブリッドインタプリタです。プログラムをするときは、にバイトコードにアセンブルし、Pythonインタプリタ *Python*マシンともばれますでできます。ライブラリの`dis`モジュールをすると、クラス、メソッド、コードオブジェクトをアセンブルすることでがめるPythonバイトコードをできます。

```
>>> 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
```

Pythonインタプリタはスタックベースであり、れしシステムをします。

Pythonアセンブリバイトコードのオペレーションコードopcodeは、スタックからのをりし、のをスタックにします。オペコードのスタックにながない、Pythonインタプリタがクラッシュするがあります。エラーメッセージはされません。

モジュールの

Pythonモジュールをアセンブルするには、まずこれを.pycファイルPythonコンパイルみにするがあります。これをうには、

```
python -m compileall <file>.py
```

にで、

```
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
```

これは、Pythonモジュールをコンパイルし、disバイトコードをします。モジュールはしてインポートされないので、できないコードでもにできます。

オンラインでディスモジュールをむ <https://riptutorial.com/ja/python/topic/1763/ディスモジュール>

108: データのコピー

Examples

浅いコピーをする

浅いコピーは、そののコピーをせずにコレクションのコピーです。

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

ディープコピーをする

ネストされたリストがあるは、ネストされたリストにもにすることがましいです。このはディープコピーと呼ばれます。

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

リストの浅いコピーをする

スライスをしてリストの浅いコピーをできます。

```
>>> l1 = [1,2,3]
>>> l2 = l1[:]      # Perform the shallow copy.
>>> l2
[1,2,3]
>>> l1 is l2
False
```

辞書をコピーする

オブジェクトにはメソッド `copy` があり `copy` 。 の浅いコピーをします。

```
>>> d1 = {1:[]}
>>> d2 = d1.copy()
>>> d1 is d2
False
```

```
>>> d1[1] is d2[1]
True
```

セットをコピーする

セットには`copy`メソッドもあり`copy`。このメソッドをして、シャローコピーをできます。

```
>>> s1 = {()}
>>> s2 = s1.copy()
>>> s1 is s2
False
>>> s2.add(3)
>>> s1
{[]}
>>> s2
{3, []}
```

オンラインでデータのコピーをむ <https://riptutorial.com/ja/python/topic/920/データのコピー>

109: データのシリアル

- `unpickled_string = pickle.loads`
- `unpickled_string = pickle.loadfile_object`
- `pickle_string = pickle.dumps(、 'cmplx'、 { 'object'、 なし})、 pickle.HIGHEST_PROTOCOL`
- `pickle.dump(、 'cmplx'、 { 'object'、 なし})、 file_object、 pickle.HIGHEST_PROTOCOL`
- `unjsoned_string = json.loads`
- `unjsoned_string = json.loadfile_object`
- `jsoned_string = json.dumps 'a'、 'b'、 'c'、 [1、 2、 3]`
- `json.dump 'a'、 'b'、 'c'、 [1、 2、 3]、 file_object`

パラメーター

パラメーター	
<code>protocol</code>	<code>pickle</code> または <code>cPickle</code> をすると、オブジェクトがシリアル/シリアルされていないメソッドです。ここで <code>pickle.HIGHEST_PROTOCOL</code> をしたいとかもしれません。これは、のメソッドをします。

JSONをする

- クロスランゲージサポート
- がめる
- `pickle`とはなり、のコードをするはありません

なぜJSONをしないのですか

- Pythonicデータをサポートしていません
- のキーは、データであってはなりません。

なぜピックル

- Pythonicタプル、、クラスをするためのらしい
- のキーはどのようなデータでもかまいません。

なぜピクルスじゃないの

- クロスサポートがありません
- のデータをロードすることはではありません

Examples

JSONをしたシリアル

JSONはクロスランゲージで、データをシリアルライズするためにくされているメソッドです

サポートされるデータ *int*、*float*、*boolean*、*string*、*list*および*dict*もつとるには -> [JSON Wiki](#)をしてください

JSONのないをすです

```
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)
```

[JSON](#)については、[JSON-Module](#)をしてください。

Pickleをしたシリアライゼーション

pickleのないをすです

```
# 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)
```

Pickleのについては、 **Pickle**をしてください。

pickleのは、セキュリティがないことをにしています。をしないデータはみまないでください。

オンラインでデータのシリアルをむ <https://riptutorial.com/ja/python/topic/3347/データのシリアル>

110: データベースアクセス

Pythonはく のなるタイプのデータベースをうことができます。これらのタイプごとにAPIがします。これらのなるAPIのをするため、PEP 249がされました。

このAPIは、データベースへのアクセスにされるPythonモジュールのをするためにされています。これをうことで、モジュールのをし、にデータベースでのがし、Pythonからのデータベースのががります。 [PEP-249](#)

Examples

MySQLdbをしたMySQL データベースへのアクセス

にうがあるのは、connectメソッドをしてデータベースへののをすることです。その、そのでするカーソルがになります。

カーソルのexecuteメソッドをしてデータベースとし、オブジェクトのcommitメソッドをしてをコミットします。

すべてがしたら、カーソルとをじることをれないでください。

Dbconnectクラスには、なものがすべてまれています。

```
import MySQLdb

class Dbconnect(object):

    def __init__(self):

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

        self.dbcursor = self.dbconnection.cursor()

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

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

データベースとするのはです。オブジェクトをしたら、executeメソッドをします。

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

ストアードプロシージャをびすは、のをします。パラメータリストはオプションであることにして

ください。

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

クエリがしたら、にのでアクセスできます。カーソルオブジェクトは、すべてのをフェッチするか、ループするジェネレータです。

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

ジェネレータをするループがな

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

データベースにをコミットするは、のようになります。

```
db.commit_db()
```

カーソルとをじるには

```
db.close_db()
```

SQLite

SQLiteは、のディスクベースのデータベースです。のデータベースサーバーをとしないので、しばしばプロトタイピングや、のに1のユーザーまたは1のユーザーがにするさなアプリケーションにされます。

```
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()
```

のコードは、`users.db`というのファイルにされているデータベースにし、ファイルがないはにします。SQLステートメントをしてデータベースとすることができます。

こののはのようになります。

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

SQLiteな

1. をしてsqliteモジュールをインポートする

```
>>> import sqlite3
```

2. モジュールをするには、にデータベースをすConnectionオブジェクトをするがあります。ここでデータはexample.dbファイルにされます

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

また、あなたはまた、なをすることができます:memory:のように、RAMにデータベースをするために

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

3. Connection Cursorたら、 Cursorオブジェクトをし、そのexecute()メソッドをびしてSQLコマンドをできます。

```
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()
```

Connectionなど

1. isolation_level

のレベルをまたはするためにされるです。コミット・モードまたはDEFERRED、IMMEDIATEまたはEXCLUSIVEいずれかのなし。

2. cursor

カーソルオブジェクトは、SQLコマンドとクエリをするためにされます。

3. `commit()`

のトランザクションをコミットします。

4. `rollback()`

の`commit()`びしにえられたをロールバックします。

5. `close()`

データベースをじます。に`commit()`はびされません。に`commit()`びせずに`close()`びすと`commit()`コミットモードでないとした、すべてののがわれます。

6. `total_changes`

データベースがかれてから、またはされたのをする。

7. `execute`、`executemany`、および`executescript`

これらは、カーソルオブジェクトのとしでされます。これはショートカットです。オブジェクトをしてこれらのをびすと、のカーソルオブジェクトがされ、するカーソルオブジェクトのメソッドがびされるためです

8. `row_factory`

このは、カーソルとのをタプルとしてけれ、のをすびしなものにできます。

```
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
```

Cursor な

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

のSQLをします。SQLをパラメータすることができますSQLリテラルのわりにプレースホルダ。sqlite3モジュールは2のプレースホルダをサポートしています? "qmark style"ときプレースホルダ:name "named style"。

```
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())
```

プログラムをSQLインジェクション [SQLインジェクション](#) にしてにするがあるため、SQLコマンドにをするために%sをしないでください。

2. executemany(sql, seq_of_parameters)

シーケンスsqlにあるすべてのパラメータシーケンスまたはマッピングにしてSQLコマンドをします。sqlite3モジュールでは、シーケンスのわりにパラメータをするイテレータをすることもできます。

```
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)
```

また、イテレータオブジェクトをパラメータとしてexecutemanyにすることもできます。このは、イテレータがすのタプルをりしします。イテレータはのタプルをさなければなりません。

```
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)

これは、にSQLをするためののなメソッドです。にCOMMITをし、パラメータとしてしたSQLスクリプトをします。

sql_scriptは、strまたはbytesインスタンスにすることができ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
    );
""")
```

のセットは、SQLのSELECTステートメントとみわけてされます。SELECTステートメントのにデータをするには、カーソルをとしてい、カーソルのfetchone()メソッドをびしてするを1つするか、fetchall()をびしてするのリストをします。

イテレータ・フォームの

```
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()

のシーケンスをすクエリセットののをフェッチします。データがそれできないは、Noneをします。

```
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)

せののセットをフェッチしサイズで、リストをします。sizeがされた、fetchmanyはのをします。ながなくなったときののリストがされます。

```
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()

せのすべてのりのをフェッチし、リストをします。

```
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)]
```

SQLiteとPythonのデータ

SQLiteはネイティブに、NULL、INTEGER、REAL、TEXT、BLOBのタイプをサポートしています。

これは、SQLからPythonにするとき、またはそののときにデータがされるです。

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

psycopg2をしたPostgreSQLデータベースアクセス

psycopg2は、でなPostgreSQLデータベースアダプタとしてもしています。のPostgreSQLアダプタのです。

なは、Python DB API 2.0のなとスレッドのすいくつかのスレッドはじをできます

データベースへののとテーブルの

```
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()
```

テーブルにデータをする

```
# 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) """)
```

データの

```
# 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])
```

のはのようになります。

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

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

そして、そこに、あなたは**psycopg2**についてるがあることをしています :)

Oracleデータベース

- cx_Oracleパッケージ - すべてのバージョンは[こちら](#)
- Oracleインスタント・クライアント - [Windows x64](#)、[Linux x64](#)

セットアップ

- cx_Oracleパッケージをのようインストールします。

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

- Oracleインスタント・クライアントをし、をの用にします。

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

の

```
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()
```

データベースのバージョンをする

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

サンプル['12', '1', '0', '2', '0']

クエリを**SELECT**

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

はPythonタプルでされます

10、'SYSADMIN'、'IT-INFRA'、7

23、「」、6

クエリを**INSERT**

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

Oracle Databaseで//をすると、はcommitがされるまでセッションでのみできます。されたデータがデータベースにコミットされると、そのデータはのユーザーやセッションでできます。

クエリをバインドをした**INSERT**

バインドをすると、をするオーバーヘッドなしに、しいでをできます。バインドはコードのをさせ、SQLインジェクションのリスクをできます。

```
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()
```

をじる

```
_db_connection.close()
```

closeメソッドはをじます。にクローズされていないは、スクリプトののににされます。

の

PEP 249によると、データベースへのは、`Connection`オブジェクトをす`connect()`コンストラクタをしてするがあります。このコンストラクタのは、データベースにします。するについては、データベースのトピックをしてください。

```
import MyDBAPI

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

このオブジェクトには4つのメソッドがあります。

1 じる

```
con.close()
```

すぐにをじます。 `Connection.__del__`メソッドがびされると、はにじられることにしてください。のトランザクションはにロールバックされます。

2 コミット

```
con.commit()
```

のトランザクションをデータベースにコミットします。

3 ロールバック

```
con.rollback()
```

のトランザクションのにロールバックします。いえれば、データベースにするコミットされていないトランザクションはすべてりされます。

4 カーソル

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

`Cursor`オブジェクトをします。これは、データベースでトランザクションをするためにされます。

sqlalchemyの

データベースにsqlalchemyをするには

```
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
```

このエンジンをうことができます例えばpandasを使ってmysqlからデータフレームをする

```
import pandas as pd

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

オンラインでデータベースアクセスをむ <https://riptutorial.com/ja/python/topic/4240/データベースアクセス>

111: デコレータ

き

デコレータはソフトウェアパターンです。それらは、サブクラスをしたり、されたのソースコードをすることなく、メソッド、またはクラスのをにします。デコレータをしくすると、プロセスでなツールになることがあります。このトピックでは、Pythonでのデコレータのとアプリケーションについてします。

- `def decorator_function(f):`は、`decorator_function`というのデコレータをします。
- `@decorator_function`
`def decorated_function():`は、デコレータをラップしています。
- `decorated_function = decorator_function(decorated_function)`これはな `@decorator_function` をう
のとじ `@decorator_function`

パラメーター

パラメータ	
f	ラップされるは、

Examples

デコレータ

デコレータは、のやメソッドのをします。をパラメータとしてけり、されたをすはデコレータとしてできます。

```
# 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.")
```

@-notationはとであるシンタックスシュガーです。

```
my_function = super_secret_function(my_function)
```

デコレータのみをするためには、これをにおくことができます。この "unsugared" は、デコレータが

としてをとり、なぜそれがのをすのかをにしています。また、をさないとがこるかをします

```
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
```

したがって、、デコレータにしいをしてします。このしいは、にうがあることをい、にのをびし、にりをします。のがけったをし、それをびすなデコレータをえてみましょう。

```
#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.
```

デコレータクラス

でべたように、デコレータは、そのをうためにのにできます。は、`my_func = decorator(my_func)`とです。しかし、`decorator`がわりにクラスのはどうなりますかはまだしますが、`my_func`が`decorator`クラスのインスタンスにきえられる`my_func`が`my_func`ます。このクラスが`__call__()`マジックメソッドをしている、であるかのように`my_func`をすることはです。

```
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.
```

クラスデコレータでされたは、タイプチェックのからはもはや「」とはみなされないことにしてください。

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

`__get__`するメソッドの、の`__get__` -methodをするがあります

```
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()
```

デコレータのに。

クラスデコレータはのにして1つのインスタスしかないなので、クラスデコレータでメソッドをすると、そのクラスのすべてのインスタスでじデコレータがされます。

```
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

```

デコレータをされたのようにせる

デコレータは、メタデータをじではないためりきます。これは、メタプログラミングをしてメタデータににアクセスするときをきこすがあります。メタデータには、のドキュメントストリングとそのもまれます。 [functools.wraps](#)は、いくつかのをラッパーにコピーすることによって、されたをののようにせます。

```
from functools import wraps
```

デコレータをラップする2つののは、のがされていることをすことでじことをしています。すでにバージョンをしていないり、クラスバージョンにバージョンをさせるはありません。

として

```

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__

```

・テスト・

クラスとして

```
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__
```

・テストのひも

をつデコレータデコレータファクトリ

デコレータはただつの、つまりされるをとります。のをすはありません。

しかし、のがしばしばまれる。そのトリックは、のをとり、デコレータをすをすることです。

デコレータ

```
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()
```

デコレータがあなたにえたいのは、Hello World

な

このようなデコレータファクトリでは、カッコでのデコレータをびすがあります。

```
@decoratorfactory # Without parentheses
```

```
def test():
    pass

test()
```

TypeErrordecoratormissing 1な 'func'

デコレータクラス

```
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()
```

きのデコレータの10、

デコレータでシングルトンクラスをする

シングルトンは、クラスのインスタンスを1つのインスタンス/オブジェクトにするパターンです。デコレータをして、クラスをのインスタンスをすか、しいインスタンスをするしないように、クラスをシングルトンとしてできます。

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

このデコレータは、どのクラスにもすることができ、クラスのインスタンスがで1つされることをします。そののびしでは、のクラスインスタンスがされます。

```
@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
```

したがって、ローカルをしてクラスインスタンスをするかどうか、またはの「インスタンス」をするかどうかはなく、にじオブジェクトをします。

デコレータをってををる

```
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()
```

オンラインでデコレータをむ <https://riptutorial.com/ja/python/topic/229/デコレータ>

112: デザインパターン

き

デザインパターンは、ソフトウェアにおけるもののなです。このドキュメントのトピックは、に Python でなデザインパターンのをすることをしています。

Examples

パターン

このデザインパターンはパターンとばれます。アルゴリズムのファミリをし、それぞれをカプセルし、それらをにするためにされます。パターンは、アルゴリズムをするクライアントからしてアルゴリズムをえることをにする。

えば、はさまざまなで「く」ことができます。は、なるのによってされるとえることができる。

```
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 functionality 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))
```

このをすると、のがされます。

```
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
```

C++やJavaなどのでは、このパターンはクラスまたはインタフェースをしてをしてされています。
。 Pythonでは `types.MethodType` をってクラスににできるをにするががあります。

デザインパターンとシングルトンパターンの

デザインパターンは、ソフトウェアにおける `commonly occurring problems` をします。デザインパターンはGoF(Gang of Four)にされましGoF(Gang of Four)そこでは、なパターンがもりされるとそののとしてされていました。

デザインパターンには4つのながあります。

1. The pattern nameは、の、その、およびを1つまたは2つのでするためにできるハンドルです。
2. The problemは、パターンをいつするかをします。
3. The solutionは、デザイン、その、およびコラボレーションをするをします。
4. The consequencesはパターンをするとトレードオフです。

デザインパターンの

1. のプロジェクトにわたってです。
2. のアーキテクチャレベルをすることができます
3. らは、やのである、をかけてされ、にされています
4. とがあります

デザインパターンは3つのカテゴリにできます。

- 1.

なパターン

2. パターン

3. パターン

Creational Pattern - オブジェクトのと、オブジェクトのをすることになっています。

Structural Pattern - クラスとオブジェクトのをし、よりきなをるようにします。

Behavioral Pattern - オブジェクトのとオブジェクトのにしています。

シングルトンパターン

それはえられたタイプの1つのオブジェクトと1つのオブジェクトだけをつメカニズムをし、グローバルなアクセスポイントをするのcreational patternです。

えば、シングルトンはデータベースでできます。ここでは、データベースオブジェクトがデータのをするようにします。

SingletonパターンをPythonでするには、Singletonクラスのインスタンスを1つだけし、じオブジェクトをびするがあります。

```
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)
```

```
('Object created', <__main__.Singleton object at 0x10a7cc310>)
('Object2 created', <__main__.Singleton object at 0x10a7cc310>)
```

C++やJavaなどのでは、このパターンは、コンストラクタをプライベートにしてオブジェクトのをメソッドをすることです。このでは、1つのオブジェクトがのびしでされ、クラスはそのじオブジェクトをします。しかし、Pythonではプライベートコンストラクタをするはありません。

パターン

のパターンもまた、Creational patternです。「factory」というのは、クラスがのタイプのオブジェクトをすることをします。オブジェクトとメソッドがけられたファクトリとしてするクラスがあります。クライアントは、のパラメータをつメソッドをびしてオブジェクトをし、ファクトリをしてクライアントにします。

```

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)

```

```

Which music you want to play Mp3 or Ogg"Ogg"
Playing .ogg music!

```

MusicFactoryは、ユーザがするにじて、 Mp3タイプまたはOggタイプのオブジェクトをするファクトリクラスです。

プロキシ

プロキシオブジェクトは、のオブジェクトへのアクセスをにするためによくされます。なビジネスロジックは、のによってされたくありません。

のアクセスのユーザーだけがリソースにアクセスできることをしたいとします。

プロキシにはをることができるユーザーだけがconsumer_serviceにアクセスできるようになります

```

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)
```

- アクセスがされたときの `ReservationService` はけています。
- たちはビジネスのデータ `date_from`、`date_to`、`reservations_count` をドメインのなユーザーとみわせてサービスしていません。
- **Consumer** `StatsService` にはパーミッションのロジックもあり `StatsService`

- プロキシインターフェイスは、にのオブジェクトとまったくじです。したがって、プロキシによってラップされたサービスをするユーザーは、プロキシのをしていませんでした。

オンラインでデザインパターンをむ <https://riptutorial.com/ja/python/topic/8056/デザインパターン>

113: デバッグ

Examples

Pythonデバッガ_pdb_をしたステップスルーデバッグ

Pythonライブラリには、`pdb`というインタラクティブなデバッグライブラリがまれています。`pdb`にはながあります。もにされるのは、プログラムを「ステップスルー」するです。

ステップスルーデバッグをすぐにするには

```
python -m pdb <my_file.py>
```

これにより、プログラムののでデバッグがします。

、デバッグのためにコードののセクションをにしたいとうでしょう。これをうには、`pdb`ライブラリをインポートし、`set_trace`をしてこののあるサンプルコードのフローをします。

```
import pdb

def divide(a, b):
    pdb.set_trace()
    return a/b
    # What's wrong with this? Hint: 2 != 3

print divide(1, 2)
```

このプログラムをすると、インタラクティブなデバッグがします。

```
python foo.py
> ~/scratch/foo.py(5)divide()
-> return a/b
(Pdb)
```

くの、このコマンドは1でされるので、1つのでコメントアウトすることができます

```
import pdf; pdb.set_trace()
```

`Pdb`プロンプトで、プロンプトをすることができます。これらのコマンドは、デバッグコマンドまたはPythonにすることができます。をするには、デバッガから`p`をうか、pythonの`print`をうことができます。

```
(Pdb) p a
1
(Pdb) print a
1
```

すべてのローカルのリストをするには

```
locals
```

みみ

これらは、デバッガののれたコマンドをとっています

```
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 is called
again)
```

デバッガはPythonをインタラクティブにすることもできます

```
-> 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]
```

のどれかがデバッガコマンドとするは、'のにをにし、デバッガコマンドをすることはできません。たとえば、カウンタに「**c**」をすると、デバッガでそのをすることがよくあります。な'**c**'コマンドはのブレークポイントまでをします。わりに'**c**'をして、のをののようにします。

```
(Pdb) !c
4
```

IPythonとipdbで

IPython または Jupyter がインストールされている、[のコマンド](#)をしてデバッガをびすことができます。


```
import ipdb
ipdb.set_trace()
```

すると、コードはしてされます

```
/home/usr/ook.py(3)<module>()
  1 import ipdb
  2 ipdb.set_trace()
----> 3 print("Hello world!")

ipdb>
```

らかに、これはコードをするがあることをします。もっとながあります

```
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
                                     color_scheme='Linux',
                                     call_pdb=1)
```

キャッチされていないがした、デバッガがびされます。

リモートデバッガ

いくつかの、のプロセスによってされるpythonコードをデバッグするがあります。この、 [rpdb](#) はです。

rpdbは、**stdin**と**stdout**をソケットハンドラにルーティングする**pdb**のラッパーです。デフォルトでは、ポート4444でデバッガをきます

```
# In the Python file you want to debug.
import rpdb
rpdb.set_trace()
```

そして、このプロセスにするにはターミナルでこれをするがあります。

```
# Call in a terminal to see the output
$ nc 127.0.0.1 4444
```

そして、あなたは**pdb** promptをるでしょう

```
> /home/usr/ook.py(3)<module>()
-> print("Hello world!")
(Pdb)
```

オンラインでデバッグをむ <https://riptutorial.com/ja/python/topic/2077/デバッグ>

114: トキンター

き

TkinterでリリースされたPythonのモナGUIグラフィカルユーザインタフェースライブラリです。このトピックでは、このライブラリとそののなについてします。

tkinterモジュールのはPython 2と3でなります。Python 2の、のようになります

```
from Tkinter import * # Capitalized
```

Python 3の、のコードをします。

```
from tkinter import * # Lowercase
```

Python 2とPython 3のするコードでは、

```
try:
    from Tkinter import *
except ImportError:
    from tkinter import *
```

または

```
from sys import version_info
if version_info.major == 2:
    from Tkinter import *
elif version_info.major == 3:
    from tkinter import *
```

は、[tkinterのドキュメント](#)をしてください。

Examples

のtkinterアプリケーション

tkinterはTk / Tcl GUIライブラリのラッパーをするGUIツールキットで、Pythonにまれています。のコードは、tkinterをしてしいウィンドウをし、ウィンドウにテキストをします。

Python 2では、とはしなるがあります。の「」をしてください。

```
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()

```

ジオメトリマネージャ

Tkinterには、`place`、`pack`、および`grid` 3つのジオメトリメカニズムがあります。

`place` マネージャはピクセルをします。

`pack` マネージャはウィジェットを4つのいずれかにします。しいウィジェットは、のウィジェットのにされます。

`grid` マネージャは、にサイズするスプレッドシートとに、ウィジェットを`grid`します。

`widget.place` のもなキーワードはのとおりです。

- `x`、ウィジェットの`x`
- `y`、ウィジェットの`y`
- `height`、ウィジェットのなさ
- `width`、ウィジェットの

`place` をしたコード

```

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()
```

パツク

`widget.pack`はのキーワードをることができます

- によってされたをめるかどうかを`expand`
- `fill`、すべてのをりつぶすNONEデフォルト、X、Y、BOTHのいずれか
- `side`、パツクするTOPデフォルト、BOTTOM、LEFT、またはRIGHT

グリッド

`widget.grid`のにもにされるキーワードはのとおりです

- `row`、ウィジェットのデフォルトでは0
- `rowspan`、ウィジェットのスパンデフォルトは1
- `column`、ウィジェットのデフォルトは0
- `columnspan`、ウィジェットがまたがるのデフォルトは1
- `sticky`グリッドセルがN、NE、E、SE、S、SW、W、NWのみわせよりもきい、ここでウィジェットをします

とのインデックスはゼロです。がし、がにします。

`grid`をしたコード

```
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()
```

`pack`と`grid`をじフレームにせてしないでくださいそうすることで、アプリケーションのデッドロ

ックがします。

オンラインでトキンターをむ <https://riptutorial.com/ja/python/topic/7574/トキンター>

115: パイオディオ

き

PyAudioは、クロスプラットフォームのオーディオI/OライブラリであるPortAudioのPythonバインディングをします。PyAudioをすると、Pythonをしてさまざまなプラットフォームでオーディオをにすることができます。PyAudioは、

- 1.pyPortAudio / fastaudioPortAudio v18 APIのPythonバインディング。
- 2.tkSnackTcl / TkとPythonのクロスプラットフォームサウンドツールキット。

stream_callbackはのスレッドメインスレッドからでびされます。stream_callbackでするのはのようになります。

1は、デバッグをするために、エラーにトレースバックをの.print

2。あるでメインスレッドにスローされるをキューにれます。

ストリームをするPortAudioに3 .return paAbort。

ブロックをしているは、Stream.readまたはStream.writeをコールしないでください。

PortAudioのコールバック

http://portaudio.com/docs/v19-doxydocs/portaudio_8h.html#a8a60fb2a5ec9cbade3f54a9c978e2710

Examples

コールバックモードオーディオI/O

```
"""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()

```

コールバックモードでは、新しいオーディオデータがなるとき、新しいオーディオデータがあるときはいつでも、PyAudioはコールバック2を呼び出します。PyAudioはスレッドでコールバックを呼び出します。これは、のシグネチャ `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` を `frame_count` フレームのオーディオデータと、/するフレームがあるかどうかをフラグをむタプルを返します。

pyaudio.Stream.start_stream 4をしてオーディオストリームのをします。これは、が **pyaudio.paComplete** をすまでコールバックを呼び出します。

ストリームをアクティブにすために、メインスレッドは `sleep5` などではいけません。

ブロッキングモードオーディオI/O

'''PyAudioのwaveファイルをしします。'''

```

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()

```

PyAudioをするには、に**pyaudio.PyAudio** 1をしてPyAudioをインスタンスエートします。これは、**portaudio**システムをします。

オーディオをまたはするには、**pyaudio.PyAudio.open** 2をして、のオーディオパラメータでストリームをきます。これは、オーディオをまたはする**pyaudio.Stream**をします。

pyaudio.Stream.writeをしてオーディオデータをストリームにきむか、**pyaudio.Stream.read**をしてストリームからオーディオデータをみんでオーディオをします。 3

「ブロッキングモード」では、**pyaudio.Stream.write**または**pyaudio.Stream.read**は、/されたすべてのフレームが/されるまでブロックします。また、オンザフライでオーディオデータをしたたり、したオーディオデータをちにしたりするには、「コールバックモード」コールバックモードのを

/するには**pyaudio.Stream.stop_stream**をし、ストリームをするには**pyaudio.Stream.close**をします。 4

に、**pyaudio.PyAudio.terminate** 5をして**portaudio**セッションをします。

オンラインでパイオディオをむ <https://riptutorial.com/ja/python/topic/10627/パイオディオ>

116: パイゲーム

き

Pygameは、Pythonでマルチメディアアプリケーション、にゲームをするためのライブラリです。
。ウェブサイトは<http://www.pygame.org/>です。

- pygame.mixer.init= 22050、サイズ= -16、チャンネル= 2、バッファ= 4096
- pygame.mixer.pre_init、サイズ、チャンネル、バッファ
- pygame.mixer.quit
- pygame.mixer.get_init
- pygame.mixer.stop
- pygame.mixer.pause
- pygame.mixer.unpause
- pygame.mixer.fadeout
- pygame.mixer.set_num_channelscount
- pygame.mixer.get_num_channels
- pygame.mixer.set_reservedcount
- pygame.mixer.find_channelforce
- pygame.mixer.get_busy

パラメーター

パラメーター	
カウント	するがあるチャンネルのののようなものをすの。
	ブール <code>False</code> か <code>True</code> かどうかをする <code>find_channel()</code> とアクティブまたはないチャンネルをさなければ <code>True</code> アクティブなチャンネルがしないまたはない <code>False</code>

Examples

pygameのインストール

pip

```
pip install pygame
```

conda

```
conda install -c tlatorre pygame=1.9.2
```

ウェブサイトからダウンロードする <http://www.pygame.org/download.shtml>

Windowsやそのオペレーティングシステムにしたインストーラをつけることができます。

プロジェクトは<http://www.pygame.org/>からもできます。

Pygameのミキサーモジュール

`pygame.mixer`モジュールはでされるのコントロールにちます `pygame` プログラムを。のところ、`mixer`モジュールには15のがあります。

`pygame.init()` で `pygame` をするとに、 `pygame.mixer` もするがあります。

のオプションをして、デフォルトをしてモジュールをします。これらのデフォルトオプションをにすることもできます。2のオプションをうことで、でしたをとってモジュールをすることが出来ます。

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

みかどうかをべるには `pygame.mixer.get_init()` します `pygame.mixer.get_init()`、 `True`は`True`、そうでないは`False`をします。を `pygame.mixer.quit()`、 `pygame.mixer.quit()` います。モジュールでサウンドをしけるは、モジュールをするがあります。

なアクション

あなたのサウンドがされているときに、 `pygame.mixer.pause()` でにすることが出来ます。サウンドのをするには、に `pygame.mixer.unpause()` します。 `pygame.mixer.fadeout()` をってサウンドのわりをフェードアウトすることも出来ます。これは、をフェードアウトさせるのにするミリであるをとります。

チャンネル

それらをサポートするのになオープンチャンネルがあるり、なだけくのをできます。デフォルトでは、8つのチャンネルがあります。チャネルのをするには、 `pygame.mixer.set_num_channels()` します。はではないです。チャンネルがすると、されたチャンネルでされているサウンドはちにします。

されているチャネルのをべるには、 `pygame.mixer.get_channels(count)` びします。は、いていないチャンネルのです。また、 `pygame.mixer.set_reserved(count)` をしてするがあるサウンドのチャンネルをすることも出来ます。もではないです。たにされたチャンネルでされるサウンドはしませ

ん。

また、 `pygame.mixer.find_channel(force)` をして、されていないチャンネルをつけることもできます。は `boolTrue` または `False` のいずれかです。アイドルのチャンネルがなく、 `force` が `False` の、 `None` をします。 `force` が `true` の、もいされていたチャンネルがされます。

オンラインでパイゲームをむ <https://riptutorial.com/ja/python/topic/8761/パイゲーム>

117: バイナリデータ

- パック fmt、v1、v2、...
- unpackfmt、buffer

Examples

のリストをバイトオブジェクトにフォーマットする

```
from struct import pack

print(pack('I3c', 123, b'a', b'b', b'c')) # b'\x00\x00\x00abc'
```

にってバイトオブジェクトをする

```
from struct import unpack

print(unpack('I3c', b'\x00\x00\x00abc')) # (123, b'a', b'b', b'c')
```

のパッキング

モジュール "**struct**" は、Python オブジェクトをしたバイトのチャンクとしてパックしたり、バイトのチャンクを Python にしたりするをします。

pack は、と 1 つのをり、2 をします。これは、がではなくバイトのチャンクであることは、のとによくています。

```
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)
```

ネイティブバイトオーダーさなバイトチャンク `\x03\x00\x00\x00\x04\x00\x05'`
バイトチャンク 3,4,5 バイトチャンク `\x03\x00\x00\x00\x04\x00\x05\x00 '`

ネットワークバイトオーダーをして、ネットワークからしたデータまたはパックデータをネットワークにすることができます。

```
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)
```

バイトチャンクネイティブバイトオーダー '\x03\x00\x04\x00\x05\x00'

バイトチャンクネットワークバイトオーダー '\x00\x03\x00\x04\x00\x05'

にしたバッファをしてしいバッファをりてるオーバーヘッドをけることで、することができます。

```
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)
```

バイトチャンク '\x03\x00\x04\x00\x05\x00\x00\x00'

バイトチャンク '\x00\x00\x03\x00\x04\x00\x05\x00'

オンラインでバイナリデータをむ <https://riptutorial.com/ja/python/topic/2978/バイナリデータ>

118: パスステートメント

- パス

なぜあなたはもしないようににいたいのですか Pythonには、`if`、`except`、`def`、`class`などのコードブロックをにすることはできないというのがあります。

しかし、にはのコードブロックがそれでなもあります。の`class`ブロックは、キャッチななど、しい`class`をできます。`except`ブロックは、しをめることがもない、「でしてください」とするもなです。イテレータがすべていをう、イテレータをする`for`のの`for`ループがです。

したがって、コードブロックにもこらないとすると、そのようなブロックには`IndentationError`しないための`pass`がです。また、に、されるだけのをむのステートメント`Ellipsis`リテラル...または、ほとんどの、ドキュメンテーションをすることができが、`pass`にがこることになっていないことをにし、はありませんにされ、なくともメモリにされる。ここには、のをったものにわれる`pass`さなきコレクションがあります。いものといものをいくつかコメントしています。

- のタイプの`Exception` `xml`をするすべてまたは

```
try:
    self.version = "Expat %d.%d.%d" % expat.version_info
except AttributeError:
    pass # unknown
```

から、ののように、のすべてのタイプを`pandas`、それはまた、おそらく、にされるべきをキャッチしているため、に、いとえられ、えは`KeyboardInterrupt`または`SystemExit`あるいは`HardwareIsOnFireError`-どのようにあなたがっていますのエラーがされたカスタムボックスではされていないため、びしのアプリケーションによってはりたいことがあります。

```
try:
    os.unlink(filename_larry)
except:
    pass
```

わりに、なくとも`except Error:`をするか、このは`except OSError:`ことが`except OSError:`。がインストールしたすべてのPythonモジュールをにすると、すべての`except ...: pass`の10がすべてのをキャッチしているので、Pythonプログラミングではまだにされています。

- しいをしなないクラスをさせる `scipy`

```
class CompileError(Exception):
    pass
```

に、クラスとしてされたクラスは、にの`__init__`や、サブクラスがしなければならないのメソッドをつことがよくあります。えは、`pebl`

```
class _BaseSubmittingController(_BaseController):
    def submit(self, tasks): pass
    def retrieve(self, deferred_results): pass
```

- `mpmath` をにすることなく、コードがいくつかのテストにしてにすることをテストする

```
for x, error in MDNewton(mp, f, (1,-2), verbose=0,
                        norm=lambda x: norm(x, inf)):
    pass
```

- クラスやでは、ブロックのものとしてされるなステートメントとして、しばしばdocstringににされています。そのような、ブロックには、"これはにもしないことをしている"とうために、例えば`pebl docstring` にえて`pass` をむかもしれません

```
class ParsingError(Exception):
    """Error encountered while parsing an ill-formed datafile."""
    pass
```

- によっては、"このメソッド/クラス/ if-block / ...はまだされていませんが、これをするになります"というプレースホルダとして`pass` がされますが、はに`Ellipsis` リテラルをみ...
python-3のみと、ののな "no-op"とをにするためです。たとえば、モデルをいストロークですると、

```
def update_agent(agent):
    ...
```

のが

```
def update_agent(agent):
    pass
```

```
def time_step(agents):
    for agent in agents:
        update_agent(agent)
```

ので`update_agent` をすることをれないでください。ただし、コードのりのがしたとおりにするかどうかをするためにいくつかのテストをしてください。このメソッドは、すべてのサブクラスですがあります。このクラスでするなはありません。または、これは`raise NotImplementedError`。このでは、このリリースではまだされていませんが、これはがどのようにされるかです

Examples

をする

```
try:
```

```
    metadata = metadata['properties']  
except KeyError:  
    pass
```

なしいをする

```
class CompileError(Exception):  
    pass
```

オンラインでパスステートメントをむ <https://riptutorial.com/ja/python/topic/6891/パスステートメント>

119: パスのと

パラメーター

パラメータ	
os.F_OK	パスのをテストするためにaccessのmodeパラメータとしてす。
Os.R_OK	pathのをテストするためにaccessのmodeパラメータにめる。
Os.W_OK	pathのきみをテストするaccessのmodeパラメータにめる。
Os.X_OK	pathがかどうかをするためのaccessのmodeパラメータにめる。

Examples

os.accessをしてチェックをする

os.accessはディレクトリがするかどうかをチェックするもっといであり、みきのためにアクセスです。

```
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 Writable
os.access(path, os.W_OK)

## Check if path is Executable
os.access(path, os.X_OK)
```

すべてのをまとめてすることもです

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.X_OK)
```

のすべては、アクセスがされているはTrue、されないはFalseします。これらはUNIXとWindowsでできます。

オンラインでパスのとのをむ <https://riptutorial.com/ja/python/topic/1262/パスのと>

120: ハッシュライブラリ

き

hashlibは、く のなるなハッシュアルゴリズムとメッセージダイジェストアルゴリズムにのインタフェースをします。 FIPSセキュアハッシュアルゴリズムSHA1、SHA224、SHA256、SHA384、およびSHA512がまれています。

Examples

のMD5ハッシュ

このモジュールは、く のなるなハッシュアルゴリズムとメッセージダイジェストアルゴリズムにのインタフェースをします。 FIPSセキュアハッシュアルゴリズムSHA1、SHA224、SHA256、SHA384、およびSHA512FIPS 180-2でとRSAのMD5アルゴリズムインターネットRFC 1321でがまれています。

それぞれのタイプのハッシュにがけられたコンストラクタメソッドが1つあります。すべてじなインタフェースをつハッシュオブジェクトをします。たとえば、 `sha1()` をしてSHA1ハッシュオブジェクトをします。

```
hash.sha1()
```

このモジュールににするハッシュアルゴリズムのコンストラクターは、 `md5()` 、 `sha1()` 、 `sha224()` 、 `sha256()` 、 `sha384()` 、 および `sha512()` です。

`update()` メソッドをして、このオブジェクトにのをできるようにになりました。いつでも、 `digest()` `hexdigest()` メソッドまたは `hexdigest()` メソッドをして、それまでにされたののダイジェストをすることができます。

```
hash.update(arg)
```

ハッシュオブジェクトをargでします。りしびしは、すべてのをしたびしとです。
`m.updatea; mupdateb`は `mupdatea + b` にする。

```
hash.digest()
```

これまでに `update` メソッドにされたのダイジェストをします。これは、 `null` バイトをむASCIIをむ `digest_size` バイトのです。

```
hash.hexdigest()
```

digestとじように、ダイジェストは2のさのとしてされ、16のみがまれます。これは、メールやのバイナリでをにするためにされます。

にをします。

```
>>> 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
```

または

```
hashlib.md5("Nobody inspects the spammish repetition").hexdigest()
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
```

OpenSSLによってされるアルゴリズム

のアルゴリズムのをのパラメータとしてるジェネリック`new()` コンストラクタもし、のハッシュや OpenSSLライブラリがするのアルゴリズムにアクセスすることができます。きコンストラクタは`new()` よりもはるかにであり、されるべきです。

OpenSSLによってされるアルゴリズムで`new()` をする

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

オンラインでハッシュライブラリをむ <https://riptutorial.com/ja/python/topic/8980/ハッシュライブラリ>

121: パフォーマンスの

まず、Pythonスクリプトのパフォーマンスをさせようとする、スクリプトのボトルネックを見つけ、がデータのなやアルゴリズムのをうことができないことにしてください。パフォーマンスのボトルネックをするには、スクリプトを**プロファイリング**することができます。に、みやすさ/デザイン/をにしてコーディングプロセスをあまりにくしやうとししないでください。Donald Knuthはについてのようにべました。

「のはすべてののであり、の97というなをれてはならない。しかし、々はそのな3でをすべきではない」

Examples

コードプロファイリング

まず、スクリプトのボトルネックを見つけ、がデータのなやアルゴリズムのをうことができないことにしてください。に、みやすさ/デザイン/をにしてコーディングプロセスをあまりにくしやうとししないでください。Donald Knuthはについてのようにべました。

「のはすべてののだが、97のさなをれてはならないが、その3のなでをしてはならない」

コードを`line_profiler`は、ライブラリである`line_profiler`と`timeit` `cProfile` またはよりい`profile` といいくつかのツールがあり`timeit`。それぞれはなるをたします。

`cProfile`はプロファイラです。びし、のり、およびイベントがされ、これらのイベントの0.001までにしてなタイミングがされます。ライブラリのドキュメント[<https://docs.python.org/2/library/profile.html>] [1]は、なユースケースをしています

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

のコードのをラップしたいは、のようにします。

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... do something ...
# ... long ...
pr.disable()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=StringIO()).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

これにより、ののようながされます。ここでは、プログラムがどこのをやしているかをく、するをできます。

```
3 function calls in 0.000 seconds
```

```
Ordered by: standard name
```

ncalls	totttime	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}

モジュール `line_profiler` [https://github.com/rkern/line_profiler] [1] は、コードを `line_profiler` するのにです。これはあらかじめスクリプトではできませんが、スニペットをとしています。は、ドキュメントをしてください。もなは、`kernprof` スクリプトをパッケージページのにつてすることです。プロファイルをするには、をでするがあります。

```
$ kernprof -l script_to_profile.py
```

`kernprof` は、`LineProfiler` のインスタンスをし、それをプロファイルとともに `__builtins__` にします。これはデコレータとしてわれるようにかれていますので、スクリプトで `@profile` プロファイルしたいを `@profile` ます。

```
@profile
def slow_function(a, b, c):
    ...
```

`kernprof` のデフォルトのは、をバイナリファイル `script_to_profile.py.lprof` です。 `kernprof` に、`[-v /- view]` オプションをすると、でフォーマットされたをすぐになることができます。それのは、をどのようにできます。

```
$ python -m line_profiler script_to_profile.py.lprof
```

に `timeit` は、コマンドラインと Python シェルのから1つのライナーまたはさなをテストするなをします。このモジュールは、リストのをうのがいか、セットをリストにするときにみみ `list()` するがいかなどのにえます。セットアップコードをするには、 `setup` キーワードまたは `-s` オプションをします。

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
```

から

```
$ python -m timeit ' "-".join(str(n) for n in range(100)) '
10000 loops, best of 3: 40.3 usec per loop
```

オンラインでパフォーマンスのをむ <https://riptutorial.com/ja/python/topic/5889/パフォーマンスの>

122: ヒープック

Examples

コレクションのと

コレクションのでのアイテムをつけるために、`heapq`モジュールには`heapq`というがあります
`nlargest`つのをします。のものはしたいアイテムの、もう1つはコレクションです。

```
import heapq

numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers)) # [200, 150, 100, 50]
```

に、コレクションのアイテムをつけるために、`nsmallest`をします。

```
print(heapq.nsmallest(4, numbers)) # [1, 2, 4, 8]
```

`nlargest`と`nsmallest`は、なデータにしてオプションのキーパラメータをります。のは、`age`プロパティをして、`people`からもいともいをします。

```
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'}]
```

コレクションのアイテム

`heap`のもいプロパティは、そののがにのであるということです `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]
```

オンラインでヒープックをむ <https://riptutorial.com/ja/python/topic/7489/ヒープック>

123: ピクルデータのシリアライゼーション

- `pickle.dump`オブジェクト、ファイル、プロトコルオブジェクトをする
- `pickle.loadfile`オブジェクトをするには
- `pickle.dumpsobject`、`protocol`オブジェクトをバイトにシリアルする
- `pickle.loadsbuffer`オブジェクトをバイトからデシリアライズする

パラメーター

パラメータ	
オブジェクト	されるオブジェクト
ファイル	オブジェクトをむオープンファイル
プロトコル	オブジェクトをするためのプロトコルオプションのパラメータ
バッファ	されたオブジェクトをむbytesオブジェクト

Pickleableタイプ

のオブジェクトはpicklableです。

- `None`、`True`、`False`
- すべてのの
- すべてのの
- `tuple` `s`、`list` `s`、`set` `s`、およびpicklableオブジェクトのみをむ`dict`
- モジュールのトップレベルでされた
- みみ
- モジュールのトップレベルでされているクラス
 - `__dict__`または`__getstate__()`びしのであるそのようなクラスのインスタンスはpicklableですは、ドキュメントをしてください。

のPythonのドキュメントについています。

pickleとセキュリティ

ピクルモジュールはではありません。インターネットなどのできないからシリアルされたデータ

をけったときにはしないでください。

Examples

Pickleをしてオブジェクトをおよびシリアルする

`pickle`モジュールは、のPythonオブジェクトをのバイトにするアルゴリズムをします。このプロセスは、オブジェクトのシリアライズともされます。その、オブジェクトをすバイトストリームをまたはし、でしてじをつしいオブジェクトをすることができます。

もなコードのために、 `dump()` と `load()` をします。

オブジェクトをするには

```
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)
```

オブジェクトをシリアルするには

```
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)
```

ピクルオブジェクトとバイトオブジェクトの

にシリアライズしてして、バイトのオブジェクトのうち、デシリアライズすることもである `dumps` と `loads` になを、 `dump` と `load` 。

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) is bytes
```

```
deserialized_data = pickle.loads(serialized_data)
# deserialized_data == data
```

ピクルされたデータをカスタマイズする

のデータは、ピクルできません。そのデータは、のだけではいけません。

pickleされるものは、`__getstate__`メソッドでできます。このメソッドは、pickleableであるものを
すがあります。

oppositeでは、`__setstate__` `__getstate__`したものを作り、オブジェクトをするがあります。

```
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
```

これをうことができます

```
>>> 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
```

ここでの、`[self.important_data]` 1つをつリストを`[self.important_data]`。これはほんのに
`__getstate__`ず、`__setstate__`がどのようにしてoppositeをうかっている、`__setstate__`は
pickleableなものを`__getstate__`ことができました。いはすべてののです `{'important_data':`
`self.important_data}`。

コンストラクタはびされませんのでは、インスタンスのことにしてください`a2`でされた
`pickle.loads`これまでびすことなく`A.__init__`ので、`A.__setstate__`すべてしなければならなかった

`__init__`、それがびされたにされているだろうし。

オンラインでピクルデータのシリアライゼーションをむ

<https://riptutorial.com/ja/python/topic/2606/ピクルデータのシリアライゼーション>

124: ピグレット

き

Pygletは、ビジュアルとサウンドにされるPythonモジュールです。のモジュールとのはありません。なについては、[pyglet.org] [1]をしてください。 [1]<http://pyglet.org>

Examples

Hello World in 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()
```

Pygletのインストール

Pythonをインストールし、コマンドラインでのようにします。

Python 2

```
pip install pyglet
```

Python 3

```
pip3 install pyglet
```

Pygletでサウンドをする

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

OpenGLのためのPygletの

```
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()
```

PygletとOpenGLをった

```
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
```

ポイントをするには、GL_POINTSをGL_LINE_LOOPにきえます。

オンラインでピグレットをむ <https://riptutorial.com/ja/python/topic/8208/ピグレット>

125: ビット

き

ビットのは、ビットレベルでバイナリをします。これらのはにであり、プロセッサによってサポートされています。これらのは、デバイスドライバ、レベルのグラフィックス、およびネットワークのにです。ここでは、Pythonのビットにするなとをします。

- $x \ll y$ ビットのシフト
- $x \gg y$ ビットのシフト
- xy ビットのAND
- $x | y$ ビットOR
- x Bitwise NOT
- $x \wedge y$ ビットのXOR

Examples

ビット AND

$\&$ はバイナリ **AND** をし、 のオペランドにビットがする、そのビットがコピーされます。つまり、

```
# 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
```

ビット OR

$|$ はバイナリ "or" をします。この、どちらのオペランドにもビットがコピーされます。つまり、

```
# 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
```

ビットの

$\wedge$ は、バイナリ₁がコピーされるバイナリ**XOR**をします。バイナリ₁は、それがちょうど**1**つのオペランドであるにのみコピーされます。これをすのは、オペランドになるにのみが₁なることです。としては、

```
# 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
```

ビットのシフト

$\ll$ は、のオペランドののがオペランドによってえられたビットだけにするビットの「シフト」をします。

```
# 2 = 0b10
2 << 2
# Out: 8
# 8 = 0b1000

bin(2 << 2)
# Out: 0b1000
```

₁ビットシフトをすることは、₂をすることとである。

```
7 << 1
# Out: 14
```

_nビットシフトをすることは、 2^{**n} によるとである。

```
3 << 4
# Out: 48
```

ビットのシフト

`>>`は、ビットごとの"シフト"をします。ここで、`n`のオペランドのは、`n`のオペランドによってえられたビットだけにします。

```
# 8 = 0b1000
8 >> 2
# Out: 2
# 2 = 0b10

bin(8 >> 2)
# Out: 0b10
```

2^n ビットシフトをすることは、 2^n によるにします。

```
36 >> 1
# Out: 18

15 >> 1
# Out: 7
```

2^n ビットシフトをすることは、 2^{2^n} とにします。

```
48 >> 4
# Out: 3

59 >> 3
# Out: 7
```

ビット NOT

`~`は、`n`のすべてのビットをします。コンピュータは**きをしている**ため、`2`をする**2**のをします。この、`n`のはゼロ0ではなくの1になります。

これは、8ビットをして**2**のをす、`0000 0000`から**0111 1111**までのパターンを0から127までののをすようにい、`1xxx xxxx`はのをすようにすることをします。

8ビットの**2**の

ビット	なし	2のの
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127

ビット	なし	2の
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
1111 1110	254	-2
1111 1111	255	-1

これは、に、 $1010\ 0110$ が $(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)$
 $(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)$
 $(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)$ の2ののをつ
 $(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)$

このようにして、のは-128 $1000\ 0000$ になります。ゼロ0は $0000\ 0000$ 、マイナス1-1は $1111\ 1111$ とされます。

しかしに、これは $\sim n = -n - 1$ します。

```
# 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
```

することができ、のにされたときに、こののなにしてください。

```
~n -> -|n+1|
```

そして、のにすると、するはのようになります。

```
~-n -> |n-1|
```

のはこののルールをしています...

```
# -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

```

めみ

すべてのビット ~ をくはのインプレースバージョンをとっています

```

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

```

オンラインでビットをむ <https://riptutorial.com/ja/python/topic/730/ビット>

126: ファイルとフォルダのI/O

き

データの、読み、またはには、オペレーティングシステムのファイルをするのは、Pythonでは簡単です。ファイルのオブジェクトの読みが簡単なのは、Pythonはファイルを開く、読みする、および書き込むためのコマンドがプロセスをします。このトピックでは、Pythonがオペレーティングシステムのファイルとどのようにインターフェイスできるかについてします。

- `file_object = openファイル[, access_mode] [, バッファリング]`

パラメーター

パラメータ	
ファイル	ファイルへのパス、またはファイルがディレクトリにあるは、ファイルのファイル
アクセスモード	ファイルのオープンをする
バッファリング	オプションのラインバッファリングにされる

クロスプラットフォームエンコーディングの

Pythonの読み込み`open()`をする、コードをクロスプラットフォームにするには、`encoding`をすることがベストプラクティスです。これは、システムのデフォルトエンコーディングがプラットフォームによって異なります。

Linuxシステムは`utf-8`をデフォルトとしてしますが、これはMACとWindowsでは必ずしもではありません。

システムのデフォルトのエンコーディングをするには、のようになっています

```
import sys
sys.getdefaultencoding()
```

のPythonインタプリタから。

したがって、しているが、あなたが持っているものとしてエンコードされていることをし、プラットフォームのをするために、にエンコードをすることがです。

```
with open('somefile.txt', 'r', encoding='UTF-8') as f:
    for line in f:
        print(line)
```

Examples

ファイルモード

`mode` パラメータでされたファイルをくことができるモードはいくつかあります。これらには、

- `'r'` - みりモード。デフォルト。それは、あなたがそれをするのではなく、ファイルをむことだけをします。このモードをする、ファイルがするがあります。
- `'w'` - きみモード。しないはしいファイルをし、それのはファイルをしてきみをします。
- `'a'` - モード。ファイルのにデータをきみます。ファイルをせず、ファイルはこのモードするがあります。
- `'rb'` - バイナリでのみりモード。これは`r`ていますが、みりはバイナリモードでされます。これもデフォルトのです。
- `'r+'` - みりモードときみモードをにいます。これにより、`r`と`w`をわなくとも、にファイルをみきすることができます。
- `'rb+'` - バイナリでのみきモード。データがバイナリであることをいて`r+`とじです。
- `'wb'` - バイナリのきみ。データをいて`w`とじですが、バイナリです。
- `'w+'` - きみモードとみりモード。`r+`とくじですが、ファイルがしないは、しいファイルがされます。そのの、ファイルはきされます。
- `'wb+'` - バイナリモードでのきみとみりモード。`w+`とじですが、データはバイナリです。
- `'ab'` - バイナリモードでします。データがバイナリで`a`をいて、`a`とです。
- `'a+'` - モードとみりモード。ファイルがしないは、しいファイルをするので、`w+`とです。そのの、ファイルポインタはファイルのにありますする。
- `'ab+'` - バイナリでモードとみりモード。データがバイナリであることをいて、`a+`とじです。

```
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 +	a	a +
む	✓	✓	x	✓	x	✓
きます	x	✓	✓	✓	✓	✓
ファイルをする	x	x	✓	✓	✓	✓
ファイルをする	x	x	✓	✓	x	x
の	わり わり					

Python 3は、`exclusive creation`ためのしいモードをしました。ってりめたりきしたり、のファイルをきしたりすることはありません。

- `'x'` - なのためにいて、ファイルがにするは`FileExistsError`ます
- `'xb'` - バイナリでなモードをく。データがバイナリであることをいて、`x`としです。
- `'x+'` - みきモード。ファイルがしないは、しいファイルをするので、`w+`とです。そのの、`FileExistsError`ます。
- `'xb+'` - きみとみりのモード。 `x+`とくじですが、データはバイナリです

	バツ	x +
む	x	✓
きます	✓	✓
ファイルをする	✓	✓
ファイルをする	x	x
の		

あなたのファイルオープンコードをもっとpythonicなでくことができます

Python 3.x 3.3

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
    # Your error handling goes here
```

Python 2では、あなたは

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
```

ファイルを1ずつむ

ファイルを1ずつりしするもなはのとおりです。

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

`readline()` は、ごとのりしにしてよりなをにします。のは、のとじです。

```
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)
```

`for`ループと`readline`をにすることはいとみなされます。

よりには、`readlines()` メソッドは、ファイルののりしなコレクションをするためにされます。

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
    for i in range(len(lines)):
        print("Line " + str(i) + ": " + line)
```

これはのようにされます。

0こんにちは

1

ファイルのなをする

ファイルI/Oのましいは、`with` キーワードをすることです。これにより、みりまたはきみがするとファイルハンドルがにじられます。

```
with open('myfile.txt') as in_file:
    content = in_file.read()

print(content)
```

または、でファイルをじるするために、あなたは`with`し、にびす`close`、を

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
in_file.close()
```

`with`をしない`with`、しないがしたにえて、ってファイルをいたままにすることがあります。

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close() # This will never be called
```

ファイルへのきみ

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

`myfile.txt`をくと、そのはのようになります

ライン1ライン2ライン3ライン4

Pythonはにをしませんので、でうがあります

```
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")
```

ライン1
2
3
4

テキストモードでいたファイルをきむときに、`os.linesep`として`os.linesep`をしないでくださいデフォルト。わりに`\n`してください。

エンコードをするは、`encoding`パラメータを`open`にするだけです。

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

また、`print`ステートメントをしてファイルにきむこともできます。このみはPython 2とPython 3ではなっていますが、そのコンセプトはにされるをりんでファイルにるといのでじです。

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
```

Python 2では、あなたは

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
```

writeをするのとはなり、printはをにします。

あるファイルのをのファイルにコピーする

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- `shutil`モジュールの

```
import shutil
shutil.copyfile(src, dst)
```

ファイルまたはパスがするかどうかをする

EAFPコーディングスタイルをし、それをこ`try`する。

```
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
```

これにより、のプロセスがファイルをチェックのでしたと、チェックがされたときにがされます。このは、のにするがあります。

- `os`モジュールの


```
import os
os.path.isfile('/path/to/some/file.txt')
```

Python 3.x 3.4

- `pathlib`を`pathlib`

```
import pathlib
path = pathlib.Path('/path/to/some/file.txt')
if path.is_file():
    ...
```

されたパスがするかどうかをするには、のEAFPプロシージャにうか、パスをにチェックします。

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## Do stuff
```

ディレクトリツリーをコピーする

```
import shutil
source='///192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

ディレクトリはすでにしてはいけません。

ファイルのに

サブディレクトリをむすべてのファイルをするには、`os.walk`をします。

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

`root_dir`は"。"にすることができます。のディレクトリからするか、またはのどのパスからするかをします。

Python 3.x 3.5

ファイルにするをしたいは、よりな[os.scandir](#)メソッドをすることもできます

```
for entry in os.scandir(path):
    if not entry.name.startswith('.') and entry.is_file():
        print(entry.name)
```

あるのでファイルをむ

だから、ファイルのののだけをしたいとしましょう

あなたは `itertools` をすることができます

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

これは、pythonのけが0からまるので、13から20までできます。したがって、1は0としてけされま
す

に `next()` キーワードをとてなをむこともできます。

また、ファイルオブジェクトをなものとしてしているは、ファイルをする2つのをさせないため、
ここでは `readline()` をしないでください

mmap をったランダムファイルアクセス

`mmap` モジュールをすると、ファイルをメモリにマッピングすることで、ファイルのにランダムに
アクセスすることができます。これは、のファイルをするわりのです。

```
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()
```

ファイルのテキストの

```
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='')
```

ファイルかであるかどうかの

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

または

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

ただし、ファイルがしないは、ともがスローされます。このようなエラーがしないようにするには、のようになります。

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

bool をします。

オンラインでファイルとフォルダのI/Oをむ <https://riptutorial.com/ja/python/topic/267/ファイルとフォルダのi-o>

127: ファイルの

き

tarball、ZIP、またはgzipファイルをまたはするには、Pythonのtarfile、zipfile、およびgzipモジュールがそれぞれされます。 Pythonのtarfileモジュールは、tarballファイルからするためのTarFile.extractall(path=".", members=None)をします。 Pythonのzipfileモジュールは、ZIPファイルをまたはするためのZipFile.extractall([path[, members[, pwd]])をします。に、Pythonのgzipモジュールは、のGzipFileクラスをします。

Examples

Python ZipFile.extractallをしてZIPファイルをする

```
file_unzip = 'filename.zip'
unzip = zipfile.ZipFile(file_unzip, 'r')
unzip.extractall()
unzip.close()
```

Python TarFile.extractallをってtarballをする

```
file_untar = 'filename.tar.gz'
untar = tarfile.TarFile(file_untar)
untar.extractall()
untar.close()
```

オンラインでファイルのをむ <https://riptutorial.com/ja/python/topic/9505/ファイルの>

128: フィルタ

- フィルタ、
- `itertools.filterfunction`、`iterable`
- `future_builtins.filter`、
- `itertools.filterfalse`、
- `itertools.filterfalsefunction`、`iterable`

パラメーター

パラメータ	
	またはびしな <code>None</code> にフィルタリングにをするのみ
りしな	フィルタリングされる

ほとんどの、[やジェネレータ](#)のは、`filter()`や`ifilter()`よりみやすく、で、です。

Examples

フィルターの

するために、`filter`にいくつかのについてシーケンスのを

```
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>
```

のないフィルタリング

functionパラメータが`None`、アイデンティティがされます。

```
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
```

チェックとしてフィルタリングする

`filter` python 3.xと`ifilter` python 2.xはジェネレータをすので、`or`や`and`のようなテストをするときににです。

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
```

100よりさいのをつけるには

```
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)
```

`next` はのこののはのをえるので、するです。

filterfalse、ifilterfalse

`itertools` -moduleには`filter`のための`filter`があり

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
```

ジェネレータ `filter` まったくじようにし `filter` が、 `False` のみをします。

```
# 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)
```

オンラインでフィルタをむ <https://riptutorial.com/ja/python/topic/201/フィルタ>

129: ブール

Examples

そして

のがであるにり、2をします。そうでなければ、ののをします。

```
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)
```

のの₁にすることができ、₀はののにすることができます。

または

いずれかののがである、のにされます。のがである、2のにされます。

```
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)

```

のの₁にすることができ、₀はののにすることができます。

ない

これは、ののをします。

```

x = True
y = not x # y = False

x = False
y = not x # y = True

```

Python はブールをにします。

```

>>> 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
```

`and`と`or`はブールをすることはされていません

`or`をすると、ののをします。そうでないは、2のをにします。 `or`とです

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

`for` and `false`のはのをし、それのはのをします。

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

な

Pythonでは、2つのバイナリをして1つのをすることができます。

```
if 3.14 < x < 3.142:
    print("x is near pi")
```

くのプログラミングでは $(3.14 < x) < 3.142$ であるが、Pythonでは $3.14 < x$ and $x < 3.142$ ようにわかります。するだろう。

オンラインでブールをむ <https://riptutorial.com/ja/python/topic/1731/ブール>

130: プラグインとエクステンションクラス

Examples

ミックスイン

オブジェクトプログラミングでは、mixinは、のクラスのクラスでなくても、のクラスであるためのメソッドをむクラスです。のクラスがmixinのメソッドにアクセスするは、によってなります。

のクラスがのをできるようにすることでのメカニズムをしますが、ののなセマンティクスはありません。ミックスインは、プログラマがなるクラスでをしたいにです。じコードをももりすのではなく、のをにミックスインにグループし、それをとするクラスにすることができます。

のミックスインをするは、ミックスインのオーダーがです。ここにながります

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(Mixin1, Mixin2):
    pass
```

このでは、MyClassとtestメソッドをびします。

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

Orderがからにあるため、はMixin1でなければなりません。これにより、スーパークラスがされたときにしないがするがあります。はのようにもっといです

```
class MyClass(Mixin2, Mixin1):
    pass
```

はのようになります。

```
>>> obj = MyClass()
>>> obj.test()
Mixin2
```

ミックスインは、カスタムプラグインをするためにできます。

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.

```

カスタマイズされたクラスをつプラグイン

Python 3.6では、[PEP 487](#)に`__init_subclass__`メソッドがされました。これは、[メタクラス](#)をせずにクラスのカスタマイズをし、[します](#)。その、このによりな[プラグイン](#)をすることができます。ここでは、[の](#)をしてこのを[します](#)。

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.")

```

```
PluginA().test()  
# Base.  
# Plugin A.  
  
PluginB().test()  
# Base.  
# Plugin B.  
  
Base.plugins  
# [__main__.PluginA, __main__.PluginB]
```

オンラインでプラグインとエクステンションクラスをむ

<https://riptutorial.com/ja/python/topic/4724/プラグインとエクステンションクラス>

131: フラスコ

き

Flaskは、Pinterest、Twilio、LinkedinなどのWebサイトをするためにされるPythonマイクロウェブフレームワークです。このトピックでは、フロントエンドとバックエンドのWebでFlaskがするさまざまなについてします。

- `@ app.route "/ urlpath"、 methods = ["GET"、 "POST"、 "DELETE"、 "PUTS"、 "HEAD"、 "OPTIONS"]`
- `@app.route "/ urlpath / <param>"、メソッド= ["GET"、 "POST"、 "DELETE"、 "PUTS"、 "HEAD"、 "OPTIONS"]`

Examples

のは、なサーバーのです。

```
# 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()
```

このスクリプトをするとすべてのがインストールされたで、ローカルサーバーがされます。ホストは、**localhost**とばれる127.0.0.1です。このサーバーは、デフォルトでポート**5000**でされます。Webサーバーにアクセスするには、Webブラウザをき、URL `localhost:5000`または127.0.0.1:5000 なしをします。、あなたのコンピュータだけがウェブサーバにアクセスできます。

`app.run()` は、 **host**、 **port**、 および**debug**の3つのパラメータがあります。ホストはデフォルトで127.0.0.1ですが、これを0.0.0.0すると、URLのプライベートIPアドレスをしてネットワークのどのデバイスからでもWebサーバーにアクセスできます。ポートはデフォルトで5000ですが、パラメータがポート80にされている、ブラウザはデフォルトでポート80をするため、ポートをするはありません。デバッグオプションにでは、プロセスではないにFlaskプロジェクトにがえられたときにサーバーがするので、このパラメータをTrueにするとちます。

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

ルーティングURL

Flaskでは、URLルーティングはにデコレータをしてわれます。これらのデコレータはルーティングやパラメータきURLのルーティングにできます。のでは、このFlaskスクリプトがWebサイト `www.example.com` しているとします。

```
@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")
def profile():
    return "You went to www.example.com/guido-van-rossum"
```

そののルートでは、`/users/`とプロフィールをつURLをすると、プロフィールをすことができます。Flaskは、すべてのユーザーに`@app.route()`をめるのはひとくでなので、URLからパラメータをるようにしています

```
@app.route("/users/<username>")
def profile(username):
    return "Welcome to the profile of " + username

cities = ["OMAHA", "MELBOURNE", "NEPAL", "STUTTART", "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
```

HTTPメソッド

2つのもなHTTPメソッドは、**GET**と**POST**です。Flaskは、されるHTTPメソッドにじて、じURLからなるコードをできます。たとえば、アカウントをつWebサービスでは、サインインページとサインインプロセスをじURLでルーティングするのがもです。ブラウザでURLをいたときとしGETリクエストはログインフォームをし、POSTリクエストログインデータをむは々にするがあります。DELETEおよびPUT HTTPメソッドをするためのルートもされます。

```
@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"
```

コードをするために、フラスコから `request` パッケージをインポートすることができます。

```
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"
```

POST リクエストからデータをするには、`request` パッケージをするがあります。

```
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"]
```

ファイルとテンプレート

`return` に HTML マークアップをするわりに、`render_template()` をすることができます

```
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)
```

これは、テンプレートファイル `about-us.html` をします。アプリケーションでこのファイルがつか
るように、ディレクトリをのでするがあります。

```
- application.py
/templates
  - about-us.html
  - login-form.html
/static
  /styles
```



```
- about-style.css
- login-style.css
/scripts
- about-script.js
- login-script.js
```

もなことは、HTMLのこれらのファイルへのようになります。

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-
style.css')}}">
```

フォルダののstylesフォルダにあるabout-style.cssをすようアプリケーションにabout-style.cssします。パスのじフォーマットは、イメージ、スタイル、スクリプト、またはファイルへのすべてのにされます。

Jinjaテンプレート

Meteor.jsとに、Flaskはフロントエンドのテンプレートサービスとされています。フラスコは、デフォルトでJinjaテンプレートをします。テンプレートをすると、HTMLファイルでやループなどのさなスニペットをできます。

テンプレートをレンダリングすると、テンプレートファイルをえるパラメータがHTMLテンプレートサービスにされます。のルートでは、ユーザーとをののからHTMLにします。

```
@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)
```

このテンプレートがレンダリングされると、render_template()からされたをできます。

profile.htmlのはprofile.htmlです。

```
<!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>

```

なるのために、のりがされます。

- `{% ... %}` はステートメントをし `{% ... %}`
- `{{ ... }}` はテンプレートがされるをす
- `{# ... #}` はコメントをし `{# ... #}` テンプレートにはまれません
- `{# ... ##}` は、のりがとしてされるべきであることをします

オブジェクト

`request` オブジェクトは、ルートにしてわれたリクエストにするをします。このオブジェクトをするには、フラスコモジュールからインポートするがあります。

```
from flask import request
```

URL パラメータ

のでは `request.method` と `request.form` がされていましたが、 `request.args` プロパティをして URL パラメータのキー/のディクショナリをすることもできます。

```

@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"

```

このコンテキストでしくするには、のURLがになりますユーザーをユーザーにきえてください

`www.example.com/api/users/guido-van-rossum?key=pa55w0Rd`

ファイルのアップロード

POSTリクエストでされたフォームにファイルアップロードがまれていた、ファイルは `request` オブジェクトをしてできます。

```
@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
```

クッキー

は、URLパラメータにしたのクッキーをむこともできる。

```
@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")
```

オンラインでフラスコをむ <https://riptutorial.com/ja/python/topic/8682/フラスコ>

132: プログラムのPython

Examples

PythonプログラムのをするときにはBig-Oとされます。があるとしましょう

```
def list_check(to_check, the_list):  
    for item in the_list:  
        if to_check == item:  
            return True  
    return False
```

これは、アイテムがリストにまれているかどうかをチェックするなです。こののさをするために、あなたは $O(n)$ とうでしょう。これは、 O としての "Order of n " がOrderとしてられていることをします。

$O(n)$ - に n はコンテナのアイテムのです

$O(k)$ - に、 k はパラメータのまたはパラメータのです

リスト

ケースパラメータがランダムにされていると

$O(1)$

コピー $O(n)$

Delスライス $O(n)$

アイテムを $O(n)$

$O(n)$

アイテムをする $O(1)$

$O(1)$

$O(n)$

スライスをする $O(k)$

スライスをする $O(n + k)$

$O(k)$

ソート $O(n \log n)$

Onk

x in sOn

mins、maxsOn

さをするO1

デキ

キューはキューです。

```
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)
```

ケースパラメータがランダムにされていると

O1

AppendleftO1

コピーOn

Ok

ExtendleftOk

ポップO1

PopleftO1

On

Ok

をする

ケースランダムにされたパラメータをの

$x \in sO1$

$s - tO1ens$

$stOminlens, \text{lent}O1ens^* \text{lent}$

ここで、 l は $maxlens1, \dots, lensn$ であり、 $s1s2s3$

$stO1entO1ent^* lenss.difference\_update$

$s.symmetric\_difference\_update(tO1ent$

$s \wedge tO1ensO1ens^* \text{lent}$

$| tO1ens + \text{lent}$

アルゴリズム...

のコンピュータのにされるいくつかのがあり、Pythonもではありません。をせずにプログラムをし、コードがきれいであるものであることをすることにしてください。したときにきすぎたりすぎたりするは、をすることができます。

80/20ルールをえておいてください。くのフィールドで、の20での80をすることができます90/10ルールともされます。コードをしようとするときはいつでも、プロファイリングをしての80がどこにあるのかをすることができます。

に「」と「」のベンチマークをします。がにをしたことをにどのようになっていますかされたコードがのバージョンよりわずかにくなったりさくなったりするは、をにしてのクリアコードになります。

なアルゴリズムとデータをするOn log nクイックソートがなに、のをソートするためにOn 2バブルソートアルゴリズムをしないでください。に、Olog nバイナリツリー、またはO1Pythonハッシュテーブルをできるには、Onがなに1000のをしないでください。

のリンクをしてください... [Python Speed Up](#)

の3つのは、にアルゴリズムのさをすためにされます。

1. Θ シータは、とのにされるため、なをします。のThetaをするなは、のをし、するをすることです。たとえば、のをえてみましょう。 Θn^3 はに Θn^2 よりもいをちます、するになく、に n_0 がするため、をすることはにです。 $3n^3 + 6n^2 + 6000 = \Theta$ えられた gn にして、 Θgn がのにっていることをす。すべての $n > n_1$ にして $0 \leq c_1 gn \leq fn \leq c_2 gn$ となるような c_1, c_2 および n_0 がする。 $= n_0\}$ は、 fn が gn のシートである、 $nn > 0$ のきなにしてに $c_1 gn$ と $c_2 g = n_0$ 。シ

ータのはまた、 f_n が n_0 より大きい n のにしてでなければならないこともとする。

2. ビッグO ビッグOは、アルゴリズムのをします。これは、からのみをします。たとえば、ソートのをえてみましょう。のは、のはのがかかります。たちは、ソートののさは $O(n^2)$ であるとうことができます。 $O(n^2)$ もをカバーすることにしてください。 Θ をしてのソートののさをすと、のとのに2つのステートメントをするがあります。

1. のソートののは $\Theta(n^2)$ です。
2. ソートののケースのさは $\Theta(n)$ です。

Big Oは、アルゴリズムのにがあるにです。くの、アルゴリズムをるだけでにをつけることができます。すべての $n \geq n_0$ にして $0 \leq f_n \leq c g_n$ となるような c および n_0 がする。

3. Ω Big Oがにをえるように、 Ω はなをする。 Ω Notation <は、アルゴリズムののがあるにです。のでしたように、アルゴリズムのののパフォーマンスはにではありません。オメガは、3つのでものいです。えられた g_n にして、のを Ωg_n です。すべての $n \geq n_0$ にして $0 \leq c g_n \leq f_n$ となるような c および n_0 がする。ここでは、じソートのをえてみましょう。ソートののさは $\Omega(n)$ とくことができますが、にのケースやにはなケースにがあるので、ソートについてはあまりなではありません。

オンラインでプログラムのPythonをむ <https://riptutorial.com/ja/python/topic/9185/プログラムのpython>

133: プロセスとスレッド

き

ほとんどのプログラムは1ずつされ、に1つのプロセスしかされません。スレッドは、のプロセスがいにしてれることをにします。のプロセッサをしたスレッドにより、プログラムはのプロセスをにできます。このトピックでは、Pythonでのスレッドのについてします。

Examples

グローバルインタープリタロック

Pythonマルチスレッドのパフォーマンスは、[グローバルインタープリタロック](#)のためにしばしばしんでいます。するに、Pythonプログラムでのスレッドをつことはできますが、CPUのにかかわらず、に1つのバイトコードしかにできません。

したがって、ネットワークアクセスなどのイベントによってがブロックされているのマルチスレッドは、にです。

```
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
```

`process`がに2かかりましたが、4つのプロセスをして2かかるをにできることにしてください。

しかし、くのなど、ながPythonコードでされているのマルチスレッドは、なをもたらず、よりもくなるがあります。

```
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

```

の、のプロセスがにのをにできるため、マルチプロセッシングはです。

```

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

```

のスレッドで

`threading.Thread`をして、のスレッドでをします。

```

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

```

のプロセスでする

のプロセスでをするには、`multiprocessing.Process`をします。インターフェイスは
`threading.Thread`とています

```

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

```

スレッドのの

すべてのスレッドがじプロセスでされているため、すべてのスレッドはじデータにアクセスできます。

ただし、のをするために、データへのアクセスをロックでするがあります。

```
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}
```

プロセスの

なるプロセスでされるコードは、デフォルトではじデータをしません。ただし、 `multiprocessing` モジュールには、のプロセスでをするためのプリミティブがまれています。

```
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
```

オンラインでプロセスとスレッドをむ <https://riptutorial.com/ja/python/topic/4110/プロセスとスレッド>

134: プロパティオブジェクト

Python 2では、プロパティのすべてのことができるように、クラスがオブジェクトからされていることをしてくださいしいスタイルのクラスにしてください。

Examples

@property デコレータの

@property デコレータは、のようにするクラスのメソッドをするためにできます。これがなの1つのは、のなルックアップとそののながなをするです。

えられたモジュール `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
```

その、

```
>>> 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
```

みりときみのプロパティに @property デコレータをする

@property をしてとのカスタムをするは、のパターンをします。

```
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:])
```

これをするには

```
>>> 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
```

プロパティオブジェクトの**getter**、**setter**、または**deleter**のオーバーライド

プロパティをつクラスからするは、クラスのプロパティオブジェクトをして、`getter`、`setter`または`deleter`プロパティの1つにしいをできます。

```
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)
```

にクラスになかったセッターやデリターをすることもできます。

デコレータなしのプロパティの

デコレータシンタックス@をけてをうのはですが、やはりしされています。デコレータなしでプロパティをできます。のPython 3.xのはこれをしてしています

```
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)
```

オンラインでプロパティオブジェクトをむ <https://riptutorial.com/ja/python/topic/2050/プロパティオブジェクト>

135: プロファイリング

Examples

IPythonの%%timeitとtimeit

ストリングconcatanationのプロファイリング

```
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
```

イテラブルとリストのループをプロファイリングする

```
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
```

timeit

ののりしのプロファイリング

```
>>> import timeit
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

タイムアウトコマンドライン

のをプロファイリングする

```
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
```

コマンドラインのline_profiler

プロファイリングするのに@profileディレクティブをつソースコード

```
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()
```

kernprofコマンドをしてごとにプロファイリングをする

```
$ 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)])

ページは、ページのにつくよりもほとんどになります。

cProfile プロファイラをする

PythonにはcProfileというプロファイラがあります。にtimeitをするよりもされます。

それはあなたののスクリプトをし、あなたのスクリプトのメソッドについてそれはあなたにえます

- `ncalls` メソッドがびされた
- `totttime` されたでやされたサブのびしをく
- `percall` びしごとにやされた。または、`totttime`を`ncalls`でった
- `cumtime` このサブおよびすべてのサブびしからまでにやされた。これは、であってもです。
- `percall` `cumtime`をプリミティブびしでった
- `filename:lineno(function)` のそれぞれのデータをする

cProfilerは、のコマンドラインをしてにびすことができます。

```
$ python -m cProfile main.py
```

されたプロファイリングされたメソッドのリストをメソッドでったでソートするには

```
$ python -m cProfile -s time main.py
```

オンラインでプロファイリングをむ <https://riptutorial.com/ja/python/topic/3818/プロファイリング>

136: マルチスレッド

き

スレッドは、のコマンドをにするのではなく、Pythonプログラムがのをにできるようにします。このトピックでは、スレッディングのについてし、スレッドのをします。

Examples

マルチスレッドの

threadingモジュールをして、しいスレッドthreadをし、するをりてることによって、しいthreading.Threadをすることができます。

```
import threading

def foo():
    print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

targetパラメータは、されるまたはびしオブジェクトをします。スレッドは、Threadオブジェクトにしてstartがびされるまで、をstartません。

スレッドの

```
my_thread.start() # prints 'Hello threading!'
```

my_threadがされしたので、startびすとRuntimeErrorがされます。スレッドをデーモンとしてしたいは、daemon=True kwargをすか、start()びすにmy_thread.daemonをTrueにすると、Threadがバックグラウンドでサイレントmy_thread.daemonデーモンとしてされます。

スレッドにする

あるきなジョブをいくつかのさなジョブにし、それらをにしたいが、それらのすべてがするのをつがあるは、Thread.join()があなたがしているメソッドです。

たとえば、ウェブサイトののページをダウンロードして1つのページにまとめたいとします。あなたはこれをするでしょう

```
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)

```

`join()` をしることができます。

カスタムスレッドクラスをする

`threading.Thread` クラスをすると、しいカスタム `Thread` クラスをサブクラス `threading.Thread` できます。サブクラスで `run` メソッドをオーバーライド `run` があります。

```

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.")

```

スレッドの

コードにはのスレッドがあり、それらのでにすることができます。

`queue` ライブラリから `Queue` をできます。

```

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()

```

キューをつプロデューサスレッドとコンシューマスレッドの

```

q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()

```

ワーカー.プールの

threadingと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)

```

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))

```

David BeazleyとBrian K. Jones O'Reillyによる*Python Cookbook*、3rd edition。 Copyright 2013 David Beazley and Brian Jones、978-1-449-34037-7。

なマルチスレッドの

このセクションには、マルチスレッドをしてされたもながいくつかまれています。

なプリンタローガー

すべてをするスレッドがされ、のにじてがされます。いいは、のがわったときに "にきまれた"がされていることです。

```

#!/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(wv[i]) - 1):
    new_line += wv[i] + ' '
    i += 1
    print len(new_line)
if new_line == '':
    return (line, '')

return (new_line, ' '.join(wv[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)

```

whileループをつなスレッド

```

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!")
```

このについて。

オンラインでマルチスレッドをむ <https://riptutorial.com/ja/python/topic/544/マルチスレッド>

137: マルチプロセッシング

Examples

2つのシンプルなプロセスの

のプロセスをするなは、々にされる2つのプロセスワーカーです。のでは、2つのプロセスがされます。

- `countUp()` は、1をカウントします。
- `countDown()` は `countDown()` 1をカウントダウンします。

```
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()
```

はのとおりで。

```
Up:    0
Down:   3
Up:    1
Up:    2
Down:   2
Up:    3
Down:   1
```

プールとマップの

```
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` は、シーンのにあるの `Workers` プロセスをするクラスであり、プログラマーにさせることができます。

`Pool(5)` は5つのプロセスでしいプールをし、`pool.map` [マップ](#)とじょうにします。が、のプロセスプールのにされたをします。

`map_async`、`apply`、および `apply_async` をしてものがられます。

[オンラインでマルチプロセッシングをむ](#) <https://riptutorial.com/ja/python/topic/3601/マルチプロセッシング>

138: ミックスイン

- クラス `ClassNameMainClass`、`Mixin1`、`Mixin2`、... は、の `ClassName`、メインクラス `MainClass` をつくらすするためにされ、ミックスイン `Mixin1`、`Mixin2` など
- `class ClassName Mixin1、MainClass、Mixin2、... 'main'` クラスはのクラスであるはありません。それと `mixin` とのにはいはありません

ミックスインをクラスにするのは、スーパークラスをするのと同じようにえます。ミックスイン `foo` でクラスのオブジェクトは、は `Foo` のインスタンスとなります `isinstance(instance, Foo)` `true` をします

Examples

します

Mixin は、クラスからていないなるクラスでできるのプロパティとメソッドです。オブジェクトプログラミングでは、なるクラスのオブジェクトにじをえるためにをします。オブジェクトのセットにらかのがある、そののをオブジェクトがするクラスにれます。

たとえば、`Car`、`Boat`、`Plane` というクラスがあるとします。これらのクラスのすべてのオブジェクトにはがあるため、の `travel` をし `travel`。このシナリオでは、すべてじなでします。ルートをし、それによってします。このをするには、`Vehicle` からすべてのクラスをさせ、そのクラスにをれることができます

```
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):
    ...
```

このコードをすると、びすことができ `travel car.travel("Montana")` ボート `boat.travel("Hawaii")` および `plane.travel("France")`

しかし、クラスでできないがあればどうしますかたとえば、`Car` にラジオやラジオで `play_song_on_station` というをするをしたいとしますが、ラジオをできる `Clock` もあります。`Car` と

`Clock`はクラス `Machine` をできます。しかし、すべてののがをできるわけではありません。 `Boat` と `Plane`はできませんなくともこのでは。では、コードをせずにどのようにできますかあなたはmixinをうことができます。 Pythonでは、クラスにミックスインをえることは、これをサブクラスのリストにするのとじくらいです

```
class Foo(main_super, mixin): ...
```

`Foo`は`main_super`すべてのプロパティとメソッドをしますが、 `mixin`のプロパティとメソッドもします。

したがって、 `Car`クラスと`Clock`クラスでラジオをするには、のから`Car`をオーバーライドして、のようにします。

```
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):
    ...
```

は`car.play_song_on_station(98.7)`と`clock.play_song_on_station(101.3)`をびすことができますが、 `boat.play_song_on_station(100.5)`

ミックスインのなは、このをつ「メイン」サブクラスをしていないにもかかわらず、それでもコードをする、くのなるオブジェクトにをできることです。ミックスインがなければ、ののようなことをするのはずっとしく、りしがかもしれません。

ミックスインのメソッドオーバーライド

ミックスインは、のプロパティやメソッドをクラスにミックスするためにされるのクラスです。くの、`mixin`クラスはおいのクラスやクラスのメソッドをオーバーライドしないので、これははありません。しかし、ミックスインのメソッドやプロパティをオーバーライドすると、Pythonではクラスがからにされるため、しないにつながるがあります。

たとえば、のクラスをる

```
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
```

この、Mixin2クラスはクラスであり、Mixin1によってされ、にBaseClassによってされます。したがって、のコードスニペットをすると

```
>>> x = MyClass()
>>> x.test()
Base
```

されるは、Baseクラスからのものです。これにより、コードのロジックにしないエラーがするがあり、そのことをしてにくがあります

オンラインでミックスインをむ <https://riptutorial.com/ja/python/topic/4359/ミックスイン>

139: メソッドオーバーライド

Examples

なメソッドのオーバーライド

PythonのなオーバーライドのをしますPython 2とPython 3とのとさのため、 [しいスタイルクラス](#) と `print with ()`。

```
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
```

`Child`クラスがされると、`Parent`クラスのメソッドをします。これは、クラスがつどのメソッドも、そのクラスもつことをします。では、`introduce`のためにされた`Child`、それがされているため、クラス`Parent`のクラスでにされていないにもわらず、`Child`。

このでは、`Child`がの`print_name`メソッドをするときにオーバーライドがします。このメソッドがされなかった、`c.print_name()`は"Parent"ます。しかし、`Child`は`print_name`の`Parent`のをオーバーライドしています。`print_name`、`c.print_name()`をびすと、"Child"というがされます。

オンラインでメソッドオーバーライドをむ <https://riptutorial.com/ja/python/topic/3131/メソッドオーバーライド>

140: メタクラス

き

メタクラスでは、しいクラスがデフォルトである`type`メタクラスをきえることで、Pythonクラスの、インスタンス、アクセスなどのでくすることができます。

アーキテクチャーをするときは、メタクラスでできるくのことを、よりシンプルなセマンティクスをしてできることもしてください。

- なはしばしばです。
- クラスデコレータは、をアドホックアプローチでクラスにみむことができます。
- Python 3.6には`__init_subclass__()`がされている`__init_subclass__()`、クラスはそのサブクラスのにできます。

Examples

メタクラス

`type`が3つのでびされると、それはメタクラスとしてし、しいインスタンスをします。しいクラス/がされます。

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__          # <type 'type'>
Dummy().__class__.__class__ # <type 'type'>
```

`type`をサブクラスしてカスタムメタクラスをすることはです。

```
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
```

さて、`type`と同じでクラスをするためにできる、しいカスタムの`mytype`メタクラスがあります。

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__          # <class '__main__.mytype'>
MyDummy().__class__.__class__ # <class '__main__.mytype'>
MyDummy.__custom_attribute__ # 2
```

`class`キーワードをしてしいクラスをすると、メタクラスはデフォルトでベースクラスについてされます。


```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

のではのクラスは、`object` 々のメタクラスがタイプであろうように、`object` である、`type`。デフォルトをきすることはですが、Python 2とPython 3のどちらをするかによってなります。

Python 2.x 2.7

なクラスレベルの`__metaclass__`をして、メタクラスをすることができます。

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy) # <class '__main__.mytype'>
```

Python 3.x 3.0

な`metaclass`キーワードは、メタクラスをします。

```
class MyDummy(metaclass=mytype):
    pass
type(MyDummy) # <class '__main__.mytype'>
```

クラスのキーワード `metaclass` をくは、すべて `metaclass` にされます。したがって、`class MyDummy(metaclass=mytype, x=2)` は、キーワードとして `x=2` を `mytype` コンストラクタに `mytype` ます。

は、[Pythonメタクラスのなを](#)んでください。

メタクラスをしたシングルトン

シングルトンは、クラスのインスタンスを1つのインスタンス/オブジェクトにするパターンです。[pythonのシングルトンデザインパターンのについては、こちらをしてください。](#)

```
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
```

メタクラスの

メタクラス

Python 2.x 2.7

```
class MyClass(object):  
    __metaclass__ = SomeMetaclass
```

Python 3.x 3.0

```
class MyClass(metaclass=SomeMetaclass):  
    pass
```

Python 2および3と six との

```
import six  
  
class MyClass(six.with_metaclass(SomeMetaclass)):  
    pass
```

メタクラスによるカスタム

メタクラスのは、クラスがビルドされるたびにがにされるか、がスローされるようにすることができます。このメタクラスはビルドされるクラスのをします。

```
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
```

あなたはのようにメタクラスをうことができます

```
class Spam(metaclass=VerboseMetaclass):  
    def eggs(self):  
        print("[insert example string here]")  
s = Spam()  
s.eggs()
```

はのようになります。

```
Creating class Spam
[insert example string here]
```

メタクラスの

メタクラスとはですか

Pythonでは、すべてがオブジェクトです。、、リスト、さらにはやクラスがオブジェクトです。すべてのオブジェクトはクラスのインスタンスです。

オブジェクト x のクラスをべるには、`type(x)` びすことができます

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

Pythonのほとんどのクラスは`type`インスタンスです。`type`もクラスです。インスタンスがクラスでもあるそのようなクラスは、メタクラスとばれます。

もなメタクラス

OK、そうPythonで1つのメタクラスすでにある`type`。のものをすることはできますか

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

これはをするものではありませんが、しいメタクラスです。`MyClass`が`SimplestMetaclass`のインスタンスになっています。

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

かをするメタクラス

クラスをするの`__new__`をびすに、かをうメタクラスは、`type`の`__new__`オーバーライドして、するクラスのいくつかのプロパティをします

```
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)
```

デフォルトのメタクラス

Pythonのすべてがオブジェクトであるといたことがあります。それはであり、すべてのオブジェクトにはクラスがあります

```
>>> type(1)
int
```

リテラル1は`int`インスタンスです。クラスをできます

```
>>> class Foo(object):
...     pass
... 
```

これでインスタンスできます

```
>>> bar = Foo()
```

`bar`のクラスはですか

```
>>> type(bar)
Foo
```

二ーす、`bar`は`Foo`インスタンスです。しかし`Foo`のクラスはですか

```
>>> type(Foo)
type
```

OK、`Foo`が`type`インスタンスです。`type`はどうですか

```
>>> type(type)
type
```

だからメタクラスはですかのところ、それはクラスのクラスのちょうどなであるふりをすることができます。テイクアウト

- すべてがPythonのオブジェクトであるため、すべてにクラスがあります
- クラスのクラスはメタクラスとされます
- デフォルトのメタクラスは`type`であり、それはこれまでのところもなメタクラスです

しかし、なぜあなたはメタクラスについて知っているべきですかあなたがメタプログラミングのようなのをしているや、クラスのをしたい、Pythonはかなり「ハック」で、メタクラスのはです。

オンラインでメタクラスをむ <https://riptutorial.com/ja/python/topic/286/メタクラス>

141: モジュールとパッケージのい

PythonパッケージをZIPファイルにれて、スクリプトのにのをすると、そのようにうことができます

```
import sys
sys.path.append("package.zip")
```

Examples

モジュール

モジュールはインポートなのPythonファイルです。モジュールをすると、のようになります。

module.py

```
def hi():
    print("Hello world!")
```

my_script.py

```
import module
module.hi()
```

で

```
>>> from module import hi
>>> hi()
# Hello world!
```

パッケージ

パッケージは、のPythonファイルまたはモジュールでされ、CまたはC++でされたライブラリもめることができます。のファイルではなく、フォルダがのようになります。

フォルダpackage

- __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!")
```

すべてのPythonパッケージには__init__.pyファイルがまれているがあります。スクリプト `import package` でパッケージをインポートすると、__init__.pyスクリプトがされ、パッケージのすべてののにアクセスできます。この、 `package.hi` および `package.woof` をすることができます。

オンラインでモジュールとパッケージのいをむ <https://riptutorial.com/ja/python/topic/3142/モジュールとパッケージのい>

142: モジュールのインポート

- `import module_name`
- `import module_name.submodule_name`
- `from module_name import *`
- `from module_name import submodule_name` [、 クラス 、 、 ...など]
- `module_name`から`import some_name new_name`
- `module_name.submodule_name`から`import class_name` [、 `function_name` 、 ...など]

モジュールをインポートすると、Pythonはこのモジュールのすべてのトップレベルコードをし、モジュールにまれるすべての、クラス、およびをします。あなたのモジュールをのにインポートしたいは、トップレベルのコードにし、モジュールがインポートされたときにさせたくないは `if __name__ == '__main__':` カプセルしてください。

Examples

モジュールのインポート

`import` ステートメントをします。

```
>>> import random
>>> print(random.randint(1, 10))
4
```

`import module` は `import module` をインポートし、 `module.name` をして、そのオブジェクト、、クラスなどをできるようにします。のでは、 `randint` をむ `random` モジュールがインポートされて `randint` ます。 `random` をインポートすることで、 `randint` で `random.randint` びすことができます。

モジュールをインポートしてのにりてることができます。

```
>>> import random as rn
>>> print(rn.randint(1, 10))
4
```

あなたのpythonファイル `main.py` が `custom.py` とじフォルダにある。のようにインポートすることができます

```
import custom
```

モジュールからをインポートすることもできます

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```


のをよりくモジュールにインポートするには、ドットを`import`キーワードのでのみします。

```
from urllib.request import urlopen
```

Pythonでは、をトップレベルからびす2つのがあります。1つは`import`あり、もう1つは`from`です。ののがあるときは、`import`をするがあります。`hello.py`ファイルと`world.py`ファイルに`function`というのじがあるとし`function`。`import`ステートメントはうまくいくでしょう。

```
from hello import function
from world import function

function() #world's function will be invoked. Not hello's
```

に、`import`はをします。

```
import hello
import world

hello.function() # exclusively hello's function will be invoked
world.function() # exclusively world's function will be invoked
```

あなたはにしているでも、プロジェクトでするがあり、じをつはありません`from`ステートメントは、

じにのインポートをうことができます。

```
>>> # 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
```

のキーワードとをみわせてすることもできます。

```
>>> 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
```

モジュールからのののインポート

なモジュールをインポートするわりに、されただけをインポートすることができます。

```
from random import randint # Syntax "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))      # Out: 5
```

Pythonインタプリタはどのリソースからまたはクラスを `import randint` すべきかを知っていなければなら `import randint`、また、 `import randint` はまたはクラスをするため、 `from random import randint` です。

ののと

```
from math import pi
print(pi)                  # Out: 3.14159265359
```

のでは、モジュールをインポートしていないため、エラーがします。

```
random.randrange(1, 10)    # works only if "import random" has been run before
```

```
NameError: name 'random' is not defined
```

Pythonインタプリタは、あなたが `random` をするのかしていません。このサンプルに `import random` をしてするがあります

```
import random
random.randrange(1, 10)
```

モジュールからすべてのをインポートする

```
from module_name import *
```

えば

```
from math import *
sqrt(2)      # instead of math.sqrt(2)
ceil(2.7)    # instead of math.ceil(2.7)
```

これにより、 `math` モジュールでされているすべてのがアンダースコアでまるのグローバルにインポートされますこれは、ライターがのみであるとすることをしています。

じのがすでにまたはインポートされているは、きされます。ほとんどの `from math import sqrt, ceil` ののみを `from math import sqrt, ceil` することがされます。

```
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
```

スターキのインポートは、モジュールレベルでのみされます。クラスまたはでそれらをしようとすると、`SyntaxError` ます。

```
def f():
    from math import *
```

そして

```
class A:
    from math import *
```

どちらもする

```
SyntaxError: import * only allowed at module level
```

__all__

モジュールには、`__all__` というのがあり、`from mymodule import *` `from mymodule import *` ときにどのをインポートするかをできます。

えられたモジュール

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

`imported_by_star` ^{れた} `from mymodule import *` `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
```

ただし、`not_imported_by_star` はにインポートできます

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
21
```

プログラムによるインポート

Python 2.x 2.7

びしをしてモジュールをインポートするには、`importlib`モジュールバージョン2.7のPythonにまわっていますをします。

```
import importlib
random = importlib.import_module("random")
```

`importlib.import_module()` はパッケージのサブモジュールもインポートします

```
collections_abc = importlib.import_module("collections.abc")
```

いバージョンのPythonでは、`imp`モジュールをしてください。

Python 2.x 2.7

`imp.find_module`および`imp.load_module`をして、プログラムによるインポートをします。

ライブラリのドキュメンテーションから

```
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()
```

プログラムでモジュールをインポートするために`__import__()`をしないでください `sys.modules`、`fromlist`など、`importlib.import_module()`があなたのためにうものとしやすいながあります。

のファイルシステムのからモジュールをインポートする

にPythonライブラリやサイドパッケージとしてみみモジュールとしてしていないモジュールをインポートするは、モジュールがつかったディレクトリへのパスを`sys.path`することでこれをうことができます `sys.path`。これは、のPythonがホストにするにです。

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

モジュールへのパスではなく、`mymodule`がつかったディレクトリへのパスをすることがです。

PEP8の

いくつかのPEP8スタイルのガイドラインがされています。

1. は々のにするがあります

```
from math import sqrt, ceil    # Not recommended
from math import sqrt         # Recommended
from math import ceil
```

2. モジュールのののようにインポートをします。

- ライブラリのインポート
- する
- ローカルアプリケーション/ライブラリのインポート

3. ワイルドカードのインポートは、のネームスペースののをくのでけてください。 `from module import *`と、コードののが `module` かどうかはです。これは `from module import *` -type `from module import *`の `from module import *`があるにはにてはまります。

4. インポートをしないでください。わりになインポートをしてください。

サブモジュールのインポート

```
from module.submodule import function
```

これは `module.submodule` から `function` を `module.submodule` ます。

`__import__`

`__import__()` をして、にのみが掛っているモジュールをインポートすることができます

```
if user_input == "os":
    os = __import__("os")

# equivalent to import os
```

このをして、モジュールへのファイルパスをすることもできます

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

モジュールのインポート

インタラクティブインタプリタをする、モジュールをリロードすることができます。これは、モジュールをしていてのバージョンをインポートしたいや、のモジュールのをでしてをにしたいにです。

モジュールを `import` してにすることはできません。

```
import math
math.pi = 3
print(math.pi)    # 3
import math
print(math.pi)    # 3
```

これは、インタプリタがインポートするすべてのモジュールをするためです。モジュールをインポートしようとする、インタプリタはそれをレジスタにれてもしません。インポートするしいは、レジスタからするをしたに `import` をすることです。

```
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
```

しかし、もっとでながあります。

Python 2

`reload` をします。

Python 2.x 2.3

```
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

Python 3

`reload` が `importlib` しました

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
```

オンラインでモジュールのインポートをむ <https://riptutorial.com/ja/python/topic/249/モジュールのインポート>

143: ユーザーメソッド

Examples

ユーザーメソッドオブジェクトの

ユーザーのメソッドオブジェクトは、クラスのをするときおそらくそのクラスのインスタンスをして、そのがユーザーのオブジェクト、バインドのユーザーメソッドオブジェクト、またはクラスメソッドオブジェクトのにされます。

```
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'>>
```

がユーザメソッドオブジェクトである、しいメソッドオブジェクトは、のメソッドオブジェクトにされているクラスとじか、クラスであるにのみされます。それのは、のメソッドオブジェクトがそのままされます。

```
# 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

```

タートルの

は、スクリプトで無限に繰り返されるユーザをにします。

```

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 function 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, and the end it really just changes how big the triangle is.
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize color again

```

オンラインでユーザーメソッドをむ <https://riptutorial.com/ja/python/topic/3965/ユーザーメソッド>

144: ユニットテスト

Pythonにはいくつかのユニットテストツールがあります。このドキュメントのトピックでは、`unittest` モジュールについてします。のテストツールには、`py.test` と `nosetests` ます。 [テストにするこのPythonのドキュメント](#) は、これらのツールのいくつかをくりげずにします。

Examples

のテスト

えば、ったがえられたときにプログラムがエラーをげる。このため、にったがあつたにエラーがスローされるようにするがあります。そのため、なをチェックするがあります。このでは、のをします。

```
class WrongInputException(Exception):  
    pass
```

これは、ったがえられたにします。のコンテキストでは、がテキストとしてにされます。

```
def convert2number(random_input):  
    try:  
        my_input = int(random_input)  
    except ValueError:  
        raise WrongInputException("Expected an integer!")  
    return my_input
```

がしたかどうかをするために、`assertRaises` をしてそのをチェックします。`assertRaises` は、の2つのでできます。

1. のびしをします。のはをと、2はびしはで、りのはこのびしにされます。
2. `with` をし `with`、タイプのみをにえます。これには、よりくのコードをできるというがありますが、のがじをするがあるため、になるがあるためしてするがあります。`self.assertRaises(WrongInputException, convert2number, "not a number")`

これは、のテストケースでされています。

```
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()
```

スローされてはならないをチェックするがあるかもしれません。ただし、がスローされたときにテストがにするため、テストはまったくないがあります。オプションをするだけで、2のテストメソッドは、スローされないがどのようにチェックされるかについてのケースをしています。には、これはをキャッチし、`fail`メソッドをしてテストに`fail`ます。

unittest.mock.create_autospecをったのモッキング

をする1つののは、オブジェクトののってオブジェクトをモックアウトする`create_autospec`をうこと`create_autospec`。をすることで、にびされるようにすることができます。

`custom_math.py`を`multiply`と

```
def multiply(a, b):  
    return a * b
```

また、`process_math.py` multiples_of

```
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
```

たちは、`multiply`し`multiply`で`multiples_of`テストをうことができます。のでは、Pythonのライブラリ`unittest`をしています、これは`pytest`や`nose`のようなのテストフレームワークでもできます。

```
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
```

unittest.TestCaseでのセットアップのテストとティアダウン

々、テストがされるためのコンテキストをするがあります。 `setUp` メソッドは、クラスのテストのにされます。 `tearDown` はすべてのテストのにされます。これらのメソッドはオプションです。 `TestCases` はなでよくわれるので、クラスの `setUp` メソッドと `tearDown` メソッドもびされるように、にこれらのメソッドで `super` をびすようにするがあります。 `TestCase` のでは、の `setUp` メソッドと `tearDown` メソッドがされているため、をせずにびすことができます。

```
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()
```

`addCleanup` +には、テストのにびされるをする `addCleanup` メソッドもあります。 `setUp` がしたにのみびされる `tearDown` とはに、 `setUp` のがしたでも `addCleanup` をしてされたがびされます。なとして、このメソッドは、テストのにされたさまざまなモックをすることでよくられます。

```
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)
```

これでクリーンアップをするもう1つのは、プログラマーがセットアップコードのクリーンアップコードを書くことができ、サブクラスが`tearDown super`をびすことをれたにあなたをすることです。

にするアサーション

が2つのなるメソッドをしてみ`unittest`でをスローすることをテストできます。

コンテキストマネージャをする

```
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)
```

これによりコンテキストマネージャのでコードがされ、したはがしなかったためテストにします。コードがしいのをさせた、テストはされます。

のアサーションをするは、したのをすることもできます。

```
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')
```

びしなをすることにより

```
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)
```

チェックするはのパラメーターでなければならず、びしなを2のパラメーターとしてすがあります。されたのパラメータは、びされるにされ、をトリガするパラメータをできます。

テストのアサーションの

Pythonには`assert`がありますが、Pythonのテストフレームワークは、テストにしたよりれたアサーションをとっています。にするがで、のデバッグモードにしません。

おそらくもなアサーションは`assertTrue`。これはのようになります

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test_basic(self):
        self.assertTrue(1 + 1 == 2)
```

これはにしますが、のを

```
self.assertTrue(1 + 1 == 3)
```

するでしょう。

`assertTrue`アサーションは、テストされたものからかのboolとしてキャストできるため、もなアサーションであるがいますが、しばしばよりいがあります。とじようにテストするときは、

```
self.assertEqual(1 + 1, 3)
```

がすると、メッセージはのようになります。

```
=====
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
```

がした、メッセージはのようになります。

```
=====
FAIL: test (__main__.TruthTest)
-----

Traceback (most recent call last):
  File "stuff.py", line 6, in test
```

```
self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
```

よりであるにはのをした。

アサーションのリストは、[のドキュメントにあります](#)。に、もににったアサーションをすることは
はいえです。したがって、のように、`1 + 1 == 2`であることを `assertEqual` は、`assertTrue`
`assertEqual` よりもするがい `assertTrue`。に、`a is None` で `a is None` する `a is None`、`assertIsNone` の
`assertEqual` をするがい `assertEqual`。

アサーションにはのがあることにもしてください。したがって、`assertEqual` はの `assertNotEqual` が
あり、`assertIsNone` はの `assertIsNotNone` ます。もう、にじてネガティブカウンターパートをすると
、よりなエラーメッセージがされます。

pytestによるユニットテスト

pytestのインストール

```
pip install pytest
```

テストをする

```
mkdir tests
touch tests/test_docker.py
```

docker_something/helpers.pyテストする

```
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)
```

テストはtest_docker.pyインポートし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
```

test_docker.pyオブジェクトのようなファイルをする

```
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
```

でpytestとパッチモンキー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
```

テストのは、 `test_docker.py` ファイルの `test_` でまるがあります。

```
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])
```

に1つつテストをする

```
py.test -k test_docker_install tests
py.test -k test_copy_file_to_docker tests
py.test -k test_docker_exec_something tests
```

`tests` フォルダのすべてのテストをします。

```
py.test -k test_ tests
```

オンラインでユニットテストをむ <https://riptutorial.com/ja/python/topic/631/ユニットテスト>

145: ランダムモジュール

- `random.seeda = None`、`version = 2`バージョンはPython 3.xでのみできます
- `random.getstate`
- `random.setstatestate`
- `random.randint`、`b`
- `random.randrange`
- `random.randrange`、`ステップ=1`
- `random.choiceseq`
- `random.shufflex`、`random = random.random`
- `random.samplepopulation`、`k`

Examples

ランダムとシーケンスシャッフル、とサンプル

```
import random
```

シャッフル

`random.shuffle()`をして、をかつけなで/ランダムすることができます。例えば`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!
```

のシーケンスからランダムなをります

```
print(random.choice(laughs))
# Out: He                # Output may vary!
```

サンプル

`choice`とに、のシーケンスからランダムなをりますが、できるはのとおります。

```
#                |--sequence--|--number--|
print(random.sample(laughs, 1)) # Take one element
# Out: ['Ho']                  # Output may vary!
```

じを2することはありません。

```
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よりもきなサンプル

ランダムなどの**randint**、**randrange**、**random**、および**uniform**

```
import random
```

randint

x と y のランダムなをしますをむ

```
random.randint(x, y)
```

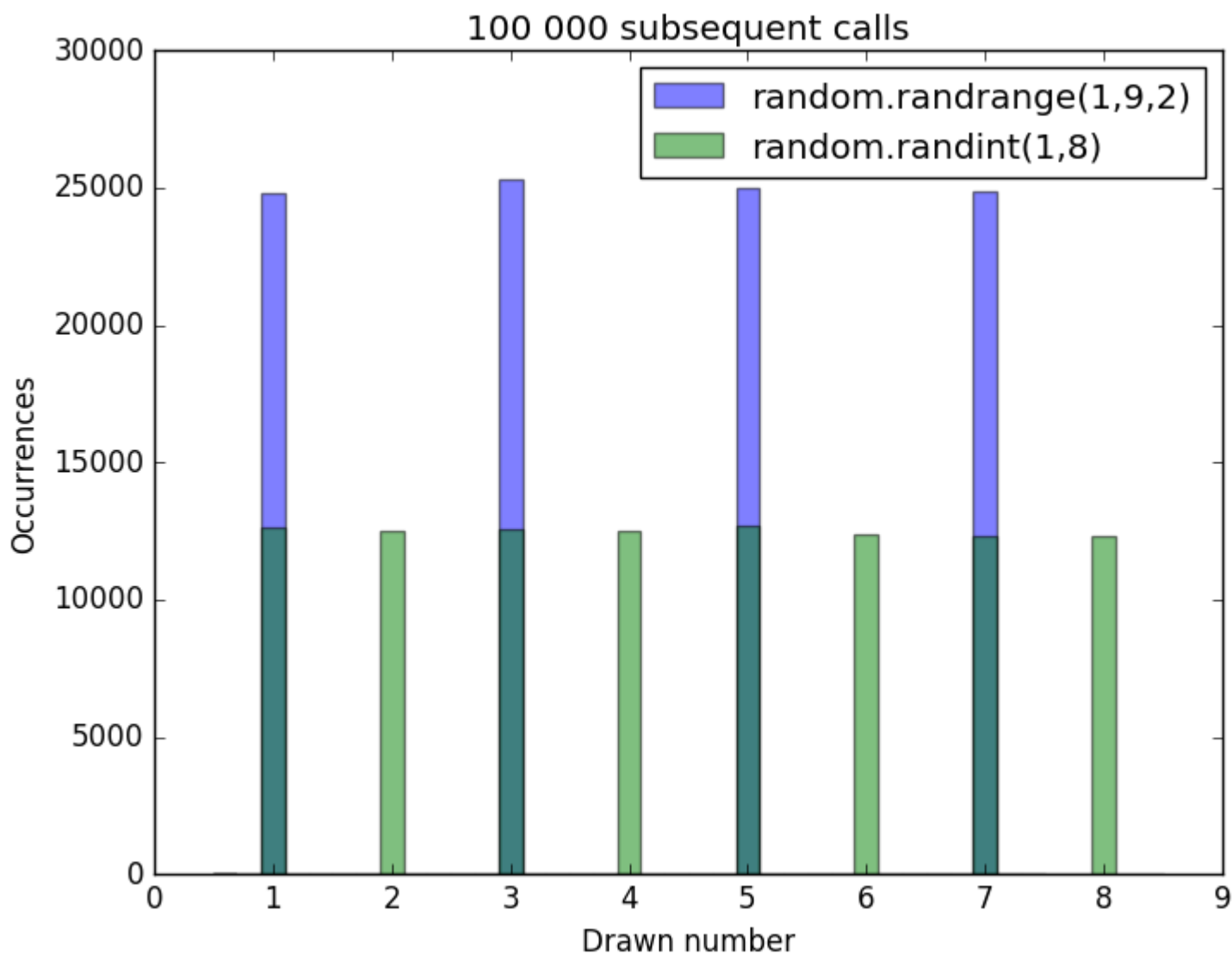
たとえば、 $_1$ から $_8$ までのをるには

```
random.randint(1, 8) # Out: 8
```

randrange

`random.randrange` じをつている `range` とはなり `random.randint`、 のはではありません。

```
random.randrange(100)      # Random integer between 0 and 99  
random.randrange(20, 50)   # Random integer between 20 and 49  
random.randrange(10, 20, 3) # Random integer between 10 and 19 with step 3 (10, 13, 16 and 19)
```



ランダム

0と1のランダムなをします。

```
random.random() # Out: 0.66486093215306317
```

x と y のランダムなをしますをむ

```
random.uniform(1, 8) # Out: 3.726062641730108
```

なシードと

のシードをすると、シリーズがされます。

```
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
```

シードをリセットすると、再び「ランダム」シーケンスがされます。

```
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
```

はされているので、これらのはに`9`と`4`です。のをただけでがじであるがないは、`getstate`と`setstate`をとってのに`getstate`こともできます。

```
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
```

シーケンスをランダムするには、`None`を`seed`します。

```
random.seed(None)
```

または、なしで`seed`メソッドをびします。

```
random.seed()
```

でなをする

デフォルトでは、PythonランダムモジュールはMersenne Twister [PRNG](#)をして[PRNG](#)をしますが、シミュレーションなどのドメインにはしてありますが、しいでセキュリティをたすことはできません。

になをするために、つはでき[SystemRandom](#)している、`os.urandom`、としてすることができる[CPRNG](#)。

それをするもなは、`SystemRandom`クラスのだけです。されるメソッドは、ランダムモジュールによってエクスポートされるメソッドにています。

```
from random import SystemRandom
```

```
secure_rand_gen = SystemRandom()
```

[0, 20] `randrange()` [0, 20]に10のintのランダムシーケンスをするには、に`randrange()`びします。

```
print([secure_rand_gen.randrange(10) for i in range(10)])  
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

えられたにランダムなをするには、`randint`をうことができます

```
print(secure_rand_gen.randint(0, 20))  
# 5
```

したがって、のすべてのについては、インターフェイスはまったくじですが、のはなジェネレーターです。

`os.urandom`して、になランダムなバイトをすることもできます。

ランダムなユーザーパスワードをする

ランダムなユーザーパスワードをするために、`string`モジュールでされるシンボルをできます。に`punctuation`の、`digits`と`digits ascii_letters`

```
from string import punctuation, ascii_letters, digits
```

に、これらすべてのシンボルを`symbols`というでみわせることができます `symbols`。

```
symbols = ascii_letters + digits + punctuation
```

これらのいずれかをして、よりないでシンボルのプールをします。

その、`random.SystemRandom`をしてパスワードをすることができます。さ10のパスワードの

```
secure_random = random.SystemRandom()  
password = "".join(secure_random.choice(symbols) for i in range(10))  
print(password) # '^@g;J?]M6e'
```

`random.choice`、`random.randint`などの`random`モジュールがすぐにできるのルーチンは、のにはさないことにしてください。

カーテンのには、`CSPRNG`のをたさないメルセンヌツイスターPRNGがわれています。したがって、に、するのパスワードをするためにそれらをしないでください。のように、に`SystemRandom`インスタンスをします。

Python 3.x 3.6

Python 3.6から、`secrets`モジュールがとなり、のながされています。

をすると、なくとも1つの、なくとも1つの、なくとも3つのをつ10ののパスワードをするには、

```
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
```

ランダムバイナリ

```
import random

probability = 0.3

if random.random() < probability:
    print("Decision with probability 0.3")
else:
    print("Decision with probability 0.7")
```

オンラインでランダムモジュールをむ <https://riptutorial.com/ja/python/topic/239/ランダムモジュール>

146: リスト

き

Python リストは、Python プログラムでくされているなデータです。それらはのでもつけられ、しばしばとばれます。それらはであり、それらをけおよびスライスすることができるシーケンス・データです。リストには、のリストオブジェクトをむさまざまなのオブジェクトをめることができます。

- `[、...]`
- `リスト[iterable]`

`list` はのタイプのりしですが、Python にするのものではありません。によっては、`set`、`tuple`、または `dictionary` をうがいでしょう

`list` は Python でにえられたです C++ の `vector<void*>` や Java の `ArrayList<Object>` ています。リンクされたリストではありません。

へのアクセスはのにわれ、にです。リストののにをするとがされますが、しばらくのは `list` のりてとコピーがになることがあります `list`。

`リスト` はリストにしています。

Examples

リストへのアクセス

Python リストはゼロインデックスされ、ののののようにします。

```
lst = [1, 2, 3, 4]
lst[0] # 1
lst[1] # 2
```

リストののインデックスにアクセスしようとする、`IndexError` ます。

```
lst[4] # IndexError: list index out of range
```

のインデックスは、リストのからえてされます。

```
lst[-1] # 4
lst[-2] # 3
lst[-5] # IndexError: list index out of range
```

これはには

```
lst[len(lst)-1] # 4
```

リストは、スライス`lst[start:end:step]`としてすることをにします。スライスのは、index `start` から `end-1` までのをむしいリストです。オプションがされた、`start` デフォルトのリストのに、リストの `end` まで、そして1に `step` します

```
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
```

これをにいて、リストののバージョンを

```
lst[::-1]    # [4, 3, 2, 1]
```

ののステップをする、インデックスはインデックスよりきくなければなりません。そうでない、はのリストになります。

```
lst[3:1:-1] # [4, 3]
```

のステップインデックスをすることは、のコードとじです。

```
reversed(lst)[0:2] # 0 = 1 -1
                  # 2 = 3 -1
```

されるインデックスは、のインデックスけでされるインデックスよりも1つなく、になります。

なスライス

リストがスライスされると、リストオブジェクトの`__getitem__()`メソッドが`slice`オブジェクトとにびされます。Pythonには、スライスオブジェクトをするためのスライスメソッドがみまれています。これをしてスライスをし、ですることができます。

```
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'
```

このクラスでは、`__getitem__`をオーバーライドすることで、オブジェクトにスライスをすることで、このを`__getitem__`にできます。

リストメソッドとサポートされる

えられたリストからめてa

```
a = [1, 2, 3, 4, 5]
```

1. append(value) - リストののしいをします。

```
# 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"]
```

append() メソッドは、リストののしいを1つだけすることにしてください。リストをのリストにすると、するリストはのリストのの1つになります。

```
# 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) - のからをしてリストをします。

```
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]
```

リストは、+とすることもできます。のリストはされません。

```
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])` - がにしたときのインデックスをします。がリストにない、`ValueError`がします。2のがされた、されたインデックスでがされます。

```
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)` - された`index`に`value`し`value`。したがって、しいは`index`をする。

```
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])` - `index`のをしてします。がなければ、リストののをしてします。

```
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)` - されたののオカレンスをします。されたがつかからない、`ValueError`がします。

```
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()` - リストをそのでし、`None`をします。

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

リストをさせるのもあります。

8. `count(value)` - リストののをえます。

```
a.count(7)
```

```
# Returns: 2
```

9. `sort()` - リストをおよびにソートし、`None`をします。

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

`sort()` メソッドの `reverse=True` フラグをしてソートすると、リストを逆にすることもできます。

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

あなたがアイテムでソートしたいは、することができます `key` キーワードを

```
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]
```

`dicts` のリストの、はじめです

```
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]
```

サブディクトでべえ

```
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]
```

attrgetterと**itemgetter**を使ってソートするがい

リストもしてすることができる**attrgetter**と**itemgetter**オペレータモジュールからします。これらは、みやすさとをさせるのにちます。はです、

```
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はインデックスを**itemgetter**こともできます。これは、タプルのインデックスについてソートするにです。

```
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) ]
```

オブジェクトのでソートするは、 **attrgetter**します。

```
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()** - リストからすべてのをします。

```
a.clear()
# a = []
```

11. `-` のリストをでけわせることで、のくのコピーからなるよりきなリストがされます。これは、リストのなどにです。

```
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]
```

あなたのリストにオブジェクトへののがまれているリストのリストなどは、[Common Pitfalls - List](#)とをしてください。

12. の `- del` キーワードとスライスをして、リストののをすることができます。

```
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. コピー

デフォルトのりて `"="` は、のリストののをしいにりてます。つまり、のとしいのがじリストオブジェクトをしています。いずれかをしてえられたは、のものにもされます。これはしばしばあなたがしたものではありません。

```
b = a
a.append(6)
# b: [1, 2, 3, 4, 5, 6]
```

リストのコピーをするは、のオプションがあります。

スライスすることができます

```
new_list = old_list[:]
```

みみのlistをうことができます

```
new_list = list(old_list)
```

な `copy.copy` をうことができます

```
import copy
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

これは、`old_list`のデータをにつけさなければならぬので、`list`よりしいです。

リストにオブジェクトがまれていて、それらもコピーしたいは、`generic copy.deepcopy`をします。

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

らかにもく、もメモリをとするですが、にはけられないものです。

Python 3.x 3.0

`copy()` - リストのいコピーをします。

```
aa = a.copy()
# aa = [1, 2, 3, 4, 5]
```

リストのさ

リストの1のさをするには、`len()`をします。

```
len(['one', 'two']) # returns 2

len(['one', [2, 3], 'four']) # returns 3, not 4
```

`len()`は、、、およびリストにたのデータでもします。

`len()`はビルトインであり、リストオブジェクトのメソッドではありません。

また、`len()`コストは $O(1)$ であり、さになくリストのさをするのにじがかかることにしてください。

リストをする

Pythonは、`for`ループをリストですること`for`サポートして`for`ます

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)

# Output: foo
# Output: bar
# Output: baz
```

あなたはにのをることができます

```
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
```

インデックスについてリストをするもう1つの

```
for i in range(0, len(my_list)):
    print(my_list[i])
#output:
>>>
foo
bar
baz
```

にリストのをすると、しないがじるがあることにしてください。

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)

# Output: foo
# Output: baz
```

こののでは、のでのアイテムをしましたが、そのために`bar`がスキップされました。

アイテムがリストにまれているかどうかの

Pythonは、がリストにあるかどうかをチェックするのがとてもです。に`in`を`in`ます。

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True

'toast' in lst
# Out: False
```

の`in`はリストよりです。にきなリストにもするがあるは、`list`を`set`にし、`set`のエ
レメントのをテストすることができます。

```
slst = set(lst)
'test' in slst
# Out: True
```

リストをする

されたリストにをす`reversed`をうことができます

```
In [3]: rev = reversed(numbers)
```

```
In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

リスト ""はこのによってされず、のと同じにとどまることにしてください。

にするには、`reverse`メソッドをすることもできます。

スライシングをして3のステップを-1にすることで、リストをすることもできますにはコピーをし、のリストはをけません

```
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]
```

リストがであるかどうかのチェック

リストのはブール`False`けられているので、`len(lst) == 0`をチェックするはありませんが、`lst`または`not lst`だけをチェックします

```
lst = []
if not lst:
    print("list is empty")

# Output: list is empty
```

リストのと

1. `list1`と`list2` をするもなはのとおりです。

```
merged = list1 + list2
```

2. `zip`はタプルのリストをします。ここで、`j`のタプルにはのシーケンスまたは`iterable`のそれぞれから`j`のがります。

```
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
```

リストのさがる、にはのものとじのだけがまれます。

```
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

```

のものとしくないさのパディングリストのNone sがitertools.zip_longest
itertools.izip_longest **パイソン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. のインデックスに

```

alist = [123, 'xyz', 'zara', 'abc']
alist.insert(3, [2009])
print("Final List :", alist)

```

```

Final List : [123, 'xyz', 'zara', 2009, 'abc']

```

すべてとすべて

`all()` をすると、iterableのすべてのがTrueであるかどうかをできます

```

nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True

```

に、 `any()` は、iterableの1つのがTrue

```

nums = [1, 1, 0, 1]
any(nums)

```

```
# True
vals = [None, None, None, False]
any(vals)
# False
```

このではリストをしています、これらのビルトインはジェネレータをむのイテレートですることとすることができます。

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

リストのをする

リストのしたをするには、リストを `set` のオブジェクトののコレクションにします。 `list` データがなは、 `list()` をしてそのをリストにすことができます。

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

リストをセットにすると、のがわれることにしてください。

リストのをするために、 `OrderedDict`

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

ネストされたリストのへのアクセス

3のリストからめる

```
alist = [[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]
```

リストのへのアクセス

```
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
```

サポートの

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Appends 11 to the end of the first list in the first list
```

ネストされたforループをしてリストをする

```
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]
```

これはリストのやのためのジェネレータとしてもできます。

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

のリストのすべてののがリストであるはありません。

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

ネストされたforループをするの。のはよりいですが、はにこれをするがあります

```
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]
```

ネストされたリストでのスライスの

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

のリスト

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

リストの

をしてリストとのシーケンスをにすることはです。どちらのオペランドもじでなければなりません。

```
[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
```

のリストがのリストのにまれる、のリストがちます。

```
[1, 10] < [1, 10, 100]
# True
```

のへのリストの

えば、`None`、リテラルなどの

```
my_list = [None] * 10
my_list = ['test'] * 10
```

なの、じはリストのすべてのを、えばセットのとじオブジェクトをします。

```
>>> 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}]
```

わりに、のなるオブジェクトでリストをするには、のようにします。

```
my_list=[{1} for _ in range(10)]
```

オンラインでリストをむ <https://riptutorial.com/ja/python/topic/209/> リスト

147: リストスライスリストのを

- `a[start:end]`は`end-1`からまります。
 - `a[start:]`はのりのからまります
 - `a[:end]`からまでの
 - `a[ステップ]`までにしないで、ステップごとに
 - `a[]`のコピー
 - [ソース](#)
-
- `lst[::-1]`はあなたにリストのコピーを`lst[::-1]`ます
 - `start`または`end`はのでもかまいません。つまり、のからではなくからえます。そう

```
a[-1]      # last item in the array
a[-2:]     # last two items in the array
a[:-2]     # everything except the last two items
```

[ソース](#)

Examples

3の「ステップ」をする

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

リストからサブリストをする

```
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']
```

スライスでリストをする

```
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]
```

スライスをしてリストをする

```
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]
```

オンラインでリストスライスリストのををむ <https://riptutorial.com/ja/python/topic/1494/リストスライス-リストのを->

148: リストの

き

リストは、のコードにすように、リストをかつなでするためのツールです.1から10のののリストを `[i ** 2 for i in range(1,11)]` のリスト `range` のダミー `i` をしてしいパターンをします。これは、よりのいでforループがなににされます。

- `[iは10で] i`なリストの
- `[iはxrangeiので]`なリストのは、ジェネレータオブジェクトでPython 2.x
- `[iが20のはi2 == 0]`フィルタき
- `[3、4、5]のyの[1、2、3のxのxはy]]`ネストされたループ
- `[iが6のiにしてi> 6 else 0]`
- `[iが4のはelse、iは20のはi2 == 0]`フィルタと
- `[3、4、5]]のyにする[[x、yの[1,2,3]のxは、]れリストの`

リストのは[PEP 202](#)でされ、Python 2.0でされました。

Examples

きリストの

[リストの](#)をえれば、をフィルタリング `if` を1つできます。

```
[<expression> for <element> in <iterable> if <condition>]
```

`<iterable><element>`について。 `<condition>`がTrueとされた、 `<expression>` は`<element>`をされたりストにします。

たとえば、これをしてのシーケンスからだけをすることができます。

```
[x for x in range(10) if x % 2 == 0]
# Out: [0, 2, 4, 6, 8]
```

[ライブデモ](#)

のコードはのものとじです

```
even_numbers = []
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)
```

```
print(even_numbers)
# Out: [0, 2, 4, 6, 8]
```

また、`e`と`c`は`x`である `[e for x in y if c]` のきリストのは、`list(filter(lambda x: c, map(lambda x: e, y)))`。

じをすにもかかわらず、のがのよりもほぼ2いというにしてください。をくにとって、[これはの](#)すてきなです。

これはかなりなっていることにしてください... `if ... else ...` にはとしてられているあなたが
することができます `<expression>` リストのの。のをえてみましょう。

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

ライブデモ

ここではフィルタではなく、リストアイテムにされるをします。

```
<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>
```

これは、のとみわせるとよりになります。

```
[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]
```

ライブデモ

あなたは、Python 2.7をしている、`xrange`よりいかもしれ`range`でしたように、いくつかので
[xrange](#) [ドキュメント](#)。

```
[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]
```

のコードはのものとじです

```
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]
```

と`if`をみわせることができます。はフィルタリングされたにします


```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

じことはだけではできませんでした

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out: ['*', '*', '*', '*', 4, '*', 6, '*', 8, '*']
```

きリストになをするフィルタ。

ネストループをしたリストの

List Comprehensionはforループをネストforできます。リストのでのネストされたforループをコーディングすることができます。また、forループごとforオプションのifテストがあります。そうするとき、forのは、のれのforをくときとじです。list comprehensionsのなはのようになります。

```
[ expression for target1 in iterable1 [if condition1]
    for target2 in iterable2 [if condition2]...
    for targetN in iterableN [if conditionN] ]
```

たとえば、のforをしてリストのリストをフラットするのコードは、

```
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]
```

は、のforをつリストのとしてにくことができます。

```
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]
```

ライブデモ

されたとリストのので、のループのforステートメントがにます。

よりコンパクトであることにえて、ネストされたものかなりくなります。

```
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

```

のびしのオーバーヘッドは140nsです。

インラインif sはにネストされ、そしてのでこりる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]

```

ライブデモ

ただし、みやすくするために、のforループをすることをすることがあります。これは、ネスティングが2レベルく、かつ/またはのがすぎるににてはまる。のネストされたループリストのがエラーをこしやすいか、またはしないをもたらすがあります。

リファクタリングフィルタとリストへのマップ

filterまたはmapは、くの、[リストにきえてください](#)。ガイド・ヴァン・ロッサムGuido Van Rossumは、[2005](#)のでこれをよくしている。

filter(P, S)は、ほとんどの、[x for x in S if P(x)]ように[x for x in S if P(x)]され、これは、もなでは、x==42をうというきながあります。ラムダはにとつてはるかにくのをしますラムダはリストのよりもいです。さらにmap(F, S)は[F(x) for x in S]map(F, S) [F(x) for x in S]ます。もちろん、くの、わりにジェネレータをすることが出来ます。

のコードは "python"とみなされ、くのpython lintersでエラーがします。

```

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

```

のもりからんだことをまえ、これらのfilterとmapをのリストにすることが出来ます。ラムダをそれぞれからする - プロセスでコードをよりみやすくする。

```

# 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)]
```

うのはさらにになります。みやすさのために、1つのマップまたはフィルタのものをにすがあります。なケースでは、これらをのリストのにきえることができます。さらに、したMapFilterプロセスについてするに、がえているプロセスのがあるのであるかをリストのからにすることができます。

```
# 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]
```

リファクタリング・クイックリファレンス

- `map(F, S) == [F(x) for x in S]`
- フィルタ

```
filter(P, S) == [x for x in S if P(x)]
```

ここで、 F と P はそれぞれをして `bool` をすです

れリストの

ネストされたリストは、ネストされたループをつリストとはなり、リストのリストです。のは、のリストのをものにすることができます。

```
#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]]
```

れのは、

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

ネストされた`for`ループをすることができる1つの。

```
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]]
```

ネストされた`for`ループのように、いかがのようにネストできるかにはありません。

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [['1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

リストので2つのリストをにする

リストのので2つのリストをにするには、`zip()`をのようになります。

```
>>> 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 ...
```

オンラインでリストのをむ <https://riptutorial.com/ja/python/topic/5265/> リストの

149: リスト

き

Pythonのリストは、でなです。それらは、リストのにをすることによって、のリストからリストをするためにすることができます。のセクションでは、これらののをし、します。

- `[x、1、2、3]` `x`のリスト、`[2、3、4]`
- `1、2、3`のジェネレータの`x`の`x + 1`は`2、3、4`
- `[xが1,2,3のはx2 == 0のはx]`フィルタをしたリストの、`[2]`
- `[x + 1ならx + 1 == 0、xは1、2、3でxの]`リスト
- `[xが0のはx + 1、xが0のは-3,4のxはxです。]`ターナリとフィルタリングによるリストの
- `{xは1、2、2、3でxのために}`はをし、`{1、2、3}`
- `{'a'1、'b'2}`Python 2.7+および`{b '、2}`をえる。3.0+のみ
- `[10,20]`の`y`の`[1、2]`の`x`のための`x + y`ネストされたループ`[11,21,12,22]`
- `[3、4、5]`の`y`の`x > 2`の、`[1、2、3のxのxの]`ループの1ののチェック
- `x > 2`の、`[3、4、5]`の`y`の`[1、2、3のxのyのxの]`2のループののチェック
- `[xがx2のxrange xx == 0]`ループされたがのにチェックされる

は、ののデータやをするです。のなは、これらをにされるにする。として、のようにないます。

- のに
- びしのとて
- ラムダので
- スタンドアロンのステートメントとして。 `[print(x) for x in range(10)]`

Examples

リストの

リストのは、`な`にをすることによってしい`list`します。もなはのとおりです。

```
[ <expression> for <element> in <iterable> ]
```

オブションのifもあります

```
[ <expression> for <element> in <iterable> if <condition> ]
```

`<iterable>` `<element>`の`<element>`は、オブションの `<condition>` がtrueとされたに`<expression>`プラグインされます。すべてののしいリストにまとめてされます。ジェネレータはれてされますが、リストはイテレータのさにしてメモリをにするイテレータをします。

されたの `list` をするには

```
squares = [x * x for x in (1, 2, 3, 4)]
# squares: [1, 4, 9, 16]
```

`for` は、`x` を `(1, 2, 3, 4)` からににします。 `x * x` のは、 `list` され `list` 。 に `list` がの `squares` にりてられます。

ここでするように ののほかに、リストのは、の `for` ループとほほじです。

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
# squares: [1, 4, 9, 16]
```

にされるは、にじてにすることができます。

```
# 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']
```

else

`else` は `List comprehension` でできますが、にしてはしてください。 `if / else` は、 `for` ループの `for` するがあります。

```
# 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']
```

これは、の、 をしていることにしてください。それは のではありません。 、 `if for...in` リストのであり、なソースからをフィルタリングするためにされます。

の`[... for x in ... for y in ...]`は、な、またはです。は、の`for`ループにうことです。

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```

これはのようになる

```
[str(x)
 for i in range(3)
  for x in foo(i)]
```

これは`[str(x) for i in range(3) for x in foo(i)]`として1にすることができます `[str(x) for i in range(3) for x in foo(i)]`

インプレースおよびそのの

リストのをするに、は`None`す *mutating*または*in-place*とばれるといをするのをしてください。

くのになは、にオブジェクトをしてオブジェクトをすだけです。インプレイスは、のオブジェクトをします。これはとばれます。のには、などのおよびがまれる。

`list.sort()`、リストをインプレースでソートしのリストをすることをします、`None`をします。したがって、それはリストののでどおりにしません

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

わりに、`sorted()`は、インプレースをソートするのではなくソートされた`list`します。

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

I/Oやインプレースなどののためのをすることがです。しかし、`for`ループはよりみやすくなります。これはPython 3でしますが、

```
[print(x) for x in (1, 2, 3)]
```

わりにをします。

```
for x in (1, 2, 3):
    print(x)
```

によっては、がリストのにしています。 `random.randrange()` は、ジェネレータのをするがありますが、いもします。さらに、 `next()` はイテレータでびすことができます。

のはではありませんが、がされるたびにがリセットされるのでがあります

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

リストの

よりなリストは、ましくないさにするか、またはみにくくなるがあります。ではあまりではありませんが、リストをのにすることはです

```
[
    x for x
    in 'foo'
    if x not in 'bar'
]
```

の

のは、リストのわりにオブジェクトをすることをいて、リストのにています。

な

Python 2.x 2.7

```
{x: x * x for x in (1, 2, 3, 4)}
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

これはちょうどのみです。

```
dict((x, x * x) for x in (1, 2, 3, 4))
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

リストのとに、たちはいくつかのをたすdictだけをするために、dictののをうことができます。

Python 2.x 2.7

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# Out: {'Exchange': 8, 'Overflow': 8}
```

または、ジェネレータをしてきします。

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
```



```
# Out: {'Exchange': 8, 'Overflow': 8}
```

からまり、をキーとのペアのフィルターとしてする

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}
```

キーとののりえ

なハッシュなをむdictをとっているしたはしないをくがあります

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

キーとをれえるには、コーディングスタイルにじていくつかのがあります。

- `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

がきいは、[itertools](#)をインポートし、`izip`または`imap`をすることをしてください。

をマージする

をみわせ、にじてれになったのをっていをきします。

```
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}
```

ただし、のアンパック [PEP 448](#) がされるがあります。

Python 3.x 3.5

```
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

は 2.0でされたリストのとはなり、Python 3.0でされ、2.7にバックポートされました。バージョン2.7では、ジェネレータと`dict()`みみをして、のをシミュレートできます。

ジェネレータの

ジェネレータのはリストとによくています。ないは、になセットをしないことです。されたジェネレータオブジェクトはりしされます。

たとえば、のコードのいをしてください。

```
# 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>
```

これらは2つのになるオブジェクトです。

- リストのは`list`オブジェクトをしますが、ジェネレータのは`generator`します。
- `generator`オブジェクトはけできず、`next`をしてをにします。

`xrange`もジェネレータオブジェクトをするので、`xrange`をします。`range`をすると、リストがされます。また、`xrange`はpython 2のそののバージョンにのみします。`xrange 3`では、`range`はジェネレータをします。については、[rangeとxrangeのいの](#)をしてください。

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

`Iterator.next()` と `xrange()` は Python 3 にはないので、`g.next()` は `next(g)` と `xrange` `Iterator.next()` `range next(g)` で `xrange` てください。

これらのをのすることはできますが、

```
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
"""
```

ユースケース

ジェネレータのはされます。つまり、ジェネレータがされたときにのみをしてします。これは、のデータセットをして、メモリのデータセットのをするをけるに、しばしばにちます。

```
for square in (x**2 for x in range(1000000)):
    #do something
```

のなは、そうするがなければ、なものをすることをけることです。このでは、は `get_objects()` リしごとにリモートAPIからされます。ものオブジェクトがし、1つずつされなければならず、パターンにするオブジェクトがするかどうかだけをるがあります。ジェネレータをすると、パターンにするオブジェクトがしたとき。

```
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
```

をする

セットのはにているリストとのが、それはセットユニークなのなしコレクションです。

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'])
```

ライブデモ

セットはけされていないことにしてください。これは、セットののがのでされたものとなるがあることをします。

set comprehensionは、2.0でされたlist comprehensionsとはなり、Python 2.7でできます。Python 2.2からPython 2.6では、`set()` をジェネレータでしてじをすることができます。

Python 2.x 2.2

```
set(x for x in range(5))
# Out: {0, 1, 2, 3, 4}
```

をしてでなをける

のリストのをしてください

```
>>> 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, ...]
```

これは、 x 1,000のにして $f(x)$ を2びすことになります。1つはをするためのびしで、もう1つは if をすることです。 $f(x)$ がになである、これはパフォーマンスにきなをえるがあります。いことに、 $f()$ をびすとがあると、くべきがられます。

わりに、のように、の [ジェネレータ](#) をすることによって、 x にしてなを1だけするがあります。

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

または、みみ [マップ](#) のをします

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

よりみやすいコードをもたらすのは、なのでは v をりしなもののリストやタプルなどに入れて、それをするです。 v が `iterable` ののになるので、は1だけされた `slow` のへのをつことになります。

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
[16, 25, 36, ...]
```

しかし、には、コードのロジックはになるがあり、みやすいようにすることがです。に、な1ライナーにしては、々の [ジェネレータ](#) がされます。

```
>>> 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, ...]
```

コンピューティングするの $f(x)$ することである `@functools.lru_cache()` Pythonの3.2 [デコレータ](#) に $f(x)$ 。このようにして、 x にする f のはに1されているので、のリストの2のびしはとじくらくになります。このではをさせるために [メモ](#) をしています。これはジェネレータをするとています。

あなたがリストをしなければならないとう

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

メソッドのいくつかはのようになります。

```
reduce(lambda x, y: x+y, l)

sum(l, [])

list(itertools.chain(*l))
```

しかし、リストのは、ものさをもたらずでしょう。

```
[item for sublist in l for item in sublist]
```

Lのサブリストがあるには、+にされるをむにづくショートカットは、なには OL^2 です。リストがなくなるにつれて、ステップでしいリストオブジェクトがされますののすべてのをコピーするがありますにいくつかのしいがされます。にするために、のをわずに、それぞれlのLサブリストがあるとします。のlにはL-1、2のlはL-2などにコピーされます。コピーは、1からLまでのxのxの、すなわち $l * L^2 / 2$ の1です。

リストのはちょうど1つのリストをし、アイテムをのからリストに1もコピーします。

タプルをむ

リストのforは、のをできます。

```
[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]
```

これはのforループとforます

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

ただし、をするがタプルのは、でむがあります。

```
[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)]
```

をってをえる

あるをたすなアイテムのをえたいときは、をってをすることができます

```
# 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
```

なはのようによります。

1. `range(1000)` をします。
2. すべての `if` のを。
3. に `1` をして、をたすアイテムにして `1` をします。
4. をたすアイテムのをするために、`1` をします。

ここでは、リストの `1` めていませんがないことにしてください。しかし、それらをしている `sum` にしています。これはジェネレーターとばれ、にしています。

リストの

データは、するににするがあるとしてみられることがよくあります。すべてのリストアイテムのタイプは、 [リスト](#) または `map()` のいずれかでできます。

```
# 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]
```

[オンラインでリストをむ](https://riptutorial.com/ja/python/topic/196/) <https://riptutorial.com/ja/python/topic/196/> リスト

150: リストによるの

Examples

とびし

としてのリストはなるのです

```
def func(myList):  
    for item in myList:  
        print(item)
```

びしにすことができます

```
func([1,2,3,5,7])  
  
1  
2  
3  
5  
7
```

またはとして

```
aList = ['a','b','c','d']  
func(aList)  
  
a  
b  
c  
d
```

オンラインでリストによるのをむ <https://riptutorial.com/ja/python/topic/7744/リストによるの>

151: リストパッキングとアンパックともいう

Examples

デストラクションのりて

では、"unpacking"をしてIterableをにできます。

としての

```
a, b = (1, 2)
print(a)
# Prints: 1
print(b)
# Prints: 2
```

iterableのさをえてしようとする、エラーがします

```
a, b, c = [1]
# Raises: ValueError: not enough values to unpack (expected 3, got 1)
```

Python 3.x 3.0

リストとしての

のを、のさのリストをアンパックすることができます。

```
head, *tail = [1, 2, 3, 4, 5]
```

ここでは、のをスカラーとしてし、のをリストとしてします。

```
print(head)
# Prints: 1
print(tail)
# Prints: [2, 3, 4, 5]
```

これはのものとです

```
l = [1, 2, 3, 4, 5]
head = l[0]
tail = l[1:]
```

また、のまたはがリストのをします。

```
a, b, *other, z = [1, 2, 3, 4, 5]
```

```
print(a, b, z, other)
# Prints: 1 2 5 [3, 4]
```

りてをするのをする

のだけになあるは、_をしてのないことをすことができます。これはまだされています_、ほとんどのとはしてしません。

```
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

りてをするにリストをする

に、で*_をしてくのができます。

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Prints: 1
```

これはにいものではありません。リストでインデックスをすることができます。のとのを1つのりてでするのがうまくいけば

```
a, *_ , b = [1, 2, 3, 4, 5]
print(a, b)
# Prints: 1 5
```

にのをすることもできます。

```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
print(a, b, c)
# Prints: 1 3 5
```

パッキングの

では、いくつかのができます。

```
def fun1(arg1, arg2, arg3):
    return (arg1, arg2, arg3)
```

3つのがえられたにのみをびしにします

```
fun1(1, 2, 3)
```

デフォルトをして、をオプションとしてできます。

```
def fun2(arg1='a', arg2='b', arg3='c'):  
    return (arg1,arg2,arg3)
```

のようなさまざまなでをびすことができます

```
fun2(1)           → (1,b,c)  
fun2(1, 2)        → (1,2,c)  
fun2(arg2=2, arg3=3) → (a,2,3)  
...
```

しかし、をしてをパックすることもできるので、 `list` や `dict` をってをりてることができます。

リストのパッキング

あなたはのリストをとっているとえてください

```
l = [1,2,3]
```

*をして、のリストをとしてをびすことができます。

```
fun1(*l)  
# Returns: (1,2,3)  
fun1(*['w', 't', 'f'])  
# Returns: ('w','t','f')
```

しかし、あなたがのにするさのリストをしなないならば

```
fun1(*['oops'])  
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3'
```

パッキングキーワードの

では、をってをパックすることもできます。 **をすると、 `dict` をパラメータとしてするように Pythonにできます。

```
d = {  
    'arg1': 1,  
    'arg2': 2,  
    'arg3': 3  
}  
fun1(**d)  
# Returns: (1, 2, 3)
```

がデフォルトをたないしかっていないときは、されるすべてのパラメータをむようにをとし、な

パラメータをたないか、エラーがします

```
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'
```

オプションのをつの、じでをとしてバックできます。

```
fun2(**d)
# Returns: (1, 2, 3)
```

しかし、デフォルトにきえられるので、をすることができます

```
fun2(**{'arg2': 2})
# Returns: ('a', 2, 'c')
```

とじように、のパラメータではないのをえることはできません。

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun2() got an unexpected keyword argument 'arg4'
```

のでは、はとオプションののをつことができ、じきをします

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

iterableだけでをびすことができます

```
fun3(*[1])
# Returns: (1, 'b', 'c')
fun3(*[1,2,3])
# Returns: (1, 2, 3)
```

またはちょうどで

```
fun3(**{'arg1':1})
# Returns: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Returns: (1, 2, 3)
```

じびしでをすることもできます。

```
fun3(*[1,2], **{'arg3':3})
# Returns: (1,2,3)
```

ただし、じにしてのをすることはできません。

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
```

```
# Raises: TypeError: fun3() got multiple values for argument 'arg2'
```

のをする

ののをけることができるをし、"コンパイル"にのををしないは、ののようにします。

```
def fun1(*args, **kwargs):  
    print(args, kwargs)
```

*argsパラメータと**kwargsパラメータは、[tuple](#)と[dict](#)にそれぞれされたなパラメータです。

```
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}
```

なPythonコードをと、のをにすときにくされていることがすぐわかります。たとえば、クラスをするはののようにします。

```
class MyString(str):  
    def __init__(self, *args, **kwarg):  
        print('Constructing MyString')  
        super(MyString, self).__init__(*args, **kwarg)
```

オンラインでリストパッキングとアンパックともいうをむ

<https://riptutorial.com/ja/python/topic/4282/リスト-パッキングとアンパックともいう->

152: リンクされたリスト

き

リンクされたリストはノードのまりであり、それぞれがとでされています。ノードは、をしてシーケンスにまとめられます。リンクリストは、リスト、スタック、キュー、などのよりなデータをするためにできます。

Examples

リンクリストの

ここでは、みみのリストオブジェクトとじメソッドのくをリンクリストとしてしています。

```
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()
```

いはみリストとよくしています。

```
l1 = LinkedList()  
l1.add('l')  
l1.add('H')  
l1.insert(1, 'e')  
l1.append('l')  
l1.append('o')  
l1.printList()
```

```
H  
e  
l  
l  
o
```

オンラインでリンクされたリストをむ <https://riptutorial.com/ja/python/topic/9299/> リンクされたリスト

153: リンクされたリスト ノード

Examples

Python でなリンクリスト ノードをく

リンクされたリストは、のいずれかです。

- Noneでされるのリスト、または
- カーゴオブジェクトとリンクリストへのをむノード。

```
#!/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")
```

オンラインでリンクされたリストノードをむ <https://riptutorial.com/ja/python/topic/6916/> リンクされたリストノード

154: ループ

き

プログラミングのもの1つとして、ループはほぼすべてのプログラミングにとってなです。ループをすると、コードののをとばれるいくつかのループをりすようにできます。このトピックでは、のタイプのループと、Pythonでループのアプリケーションをするについてします。

- while <ブール>
- <iterable>の<variable>にして
- の<variable><number>の
- の<><>、<>
- の<variable><>、<>、<ステップサイズ>
- は、<iterable>で<variable>インデックスをつ
- zip<iterable1>、<iterable2>の<variable1>、<variable2>の

パラメーター

パラメータ	
ブール	ブールのコンテキストでできる、たとえば <code>x < 10</code>
	<code>iterable</code> からの
りしな	をするもの

Examples

リストの

あなたがすることができ、リストをするfor

```
for x in ['one', 'two', 'three', 'four']:
    print(x)
```

これはリストのをします

```
one
two
three
four
```

rangeは、forループでよくわれるをします。

```
for x in range(1, 6):
    print(x)
```

は、python >= 3のなシーケンスと、python <= 2のリストになります。どちらも、forループをしてループすることができます。

```
1
2
3
4
5
```

あなたは、リストのもののループにしたいと、ののインデックスをとっているは、Pythonのすることができます `enumerate` を

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, ':', item)
```

`enumerate` はタプルをし、タプルは `index` と `item` リストののにされます。のループがされます

```
(0, 'one')
(1, 'two')
(2, 'three')
(3, 'four')
```

`map` と `lambda` をつたでリストをりします。つまり、リストののにラムダをします。

```
x = map(lambda e: e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

NBPython 3.x `map` ではリストのわりにイテレータがされるので、リストがなには、`print(list(x))` をキャストしなければなりません <http://www.riptutorial.com/python/example/8186/map--http://www.riptutorial.com/python/topic/809/incompatibilities-moving-from-python-2-to-python-3> をしてください。

Forループ

`for` ループは、`list` や `dict` などのアイテムのコレクションをりしし、コレクションのをむコードブロックをします。

```
for i in [0, 1, 2, 3, 4]:
    print(i)
```

の `for` ループはのリストをりします。

は、`i` のをリストののにします。したがって、それがあろう `i` で、`i`、`i` はのようになる、

```
0
1
2
3
4
```

`range` は、`5`までの数をすであるため、`for`ループでできます。

```
for i in range(5):
    print(i)
```

の`for`ループとまったくじ`for`します。ここでは`0`からえての`5`つのですので、`5`はされません。

なオブジェクトとイテレータ

`for`ループは、`__getitem__`または`__iter__`をするオブジェクトであるオブジェクトにしてをできます。`__iter__`は、をします。は、ののにアクセスするためにされる`next`をつオブジェクトです。

ループでしける

`break` ステートメント

`break`ステートメントがループでされると、フローはすぐにループから "れ"ます。

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

ループは、`break`ステートメントのにされません。`break`は、ループでのみにされることにしてください。の`break`ステートメントをして、そのをびしたループをすることはできません。

のコマンドをすると、`break`がたされてループがしたときに、`4`まですべてのがされます。

```
0
1
2
3
4
Breaking from loop
```

`break`は、`for`ループ`for`もできます。これは、Pythonによってされるのループです。

```
for i in (0, 1, 2, 3, 4):
```

```
print(i)
if i == 2:
    break
```

このループをすると、

```
0
1
2
```

ループがしているので、3と4はされません。

ループに `else` がある、 `break` によってループがしたときにはされません。

`continue`

`continue` は、のブロックのりのをバイパスしてループののりしにスキップしますが、ループをします。 `break` とに、 `continue` はループにしかれません

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)
```

```
0
1
3
5
```

2と4はされないことにしてください。これは、 `i == 2` または `i == 4` ときに `print(i)` をするのではなく、 `continue` がのりしにむためです。

ネストループ

`break` して `continue` するのは、レベルのループでのみします。のは、の `for` ループからけすだけで、の `while` ループからけすことはありません。

```
while True:
    for i in range(1,5):
        if i == 2:
            break    # Will only break out of the inner loop!
```

Pythonには、にのレベルのループからけるがありません。もしこのるいがならば、1つのループをにリファクタリングし、 `break` を `return` きえることができます。

の `return` を `break` としてする

`return` は、のにくコードをすることなく、からします。

にループがある、そのループのからの`return`をすることは、ループのりのコードがされない`break`することとじです ループののコードはされません。

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

ネストされたループがある、`return`はすべてのループをします。

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

されます

```
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
```

`else`をつループ

`for`と`while`ステートメントループは、オプションで`else`をつことができますには、このはにまれです。

`else`は、`for`ループがするまでするか、`while`ループがそのが`false`になってしたにのみされます。

```
for i in range(3):
    print(i)
else:
    print('done')

i = 0
while i < 3:
    print(i)
    i += 1
else:
    print('done')
```

```
0
1
2
done
```

ループがの**break**またはをさせることによって、**else**はされません。

```
for i in range(2):
    print(i)
    if i == 1:
        break
else:
    print('done')
```

```
0
1
```

のほとんどのプログラミングには、ループのこのオプションの**else**があり**else**。に**else**キーワードのは、しばしばしているとされます。

このようなもののコンセプトはDonald Knuthにされてい**else**プログラミングのやレベルのアセンブリから**if**と**goto**についてループをきすと、**else**キーワードのがになります。

例えば

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
```

のものとです。

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>

<<end>>:
```

これらは、それぞれに**else**をけるとです。

例えば

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```


のものとです。

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

`else`をつ`for`ループもにできます。には、なオブジェクトまたはシーケンスにまだいくつかのり
がある`True`のままであるループがあります。

なぜこのなをうのだろうか

`for...else`のなは、のなです

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

この`else`をさせることをなくするために、`else`を「していない」または「つからない」とえ
ることができます。

これにするいくつかのは、[\[Python-ideas\]のfor ... elseスレッドの](#)にあります。なぜ`for`と`while`ル
ープのろに '`else`'をしますか、およびそのループにする

のりし

のをえてみましょう。

```
d = {"a": 1, "b": 2, "c": 3}
```

そのキーをするには、のものをできます。

```
for key in d:
    print(key)
```

```
"a"
```

```
"b"  
"c"
```

これはのようになります。

```
for key in d.keys():  
    print(key)
```

またはPython 2

```
for key in d.iterkeys():  
    print(key)
```

をするには、のようになります。

```
for value in d.values():  
    print(value)
```

```
1  
2  
3
```

キーとをするには、のようになります。

```
for key, value in d.items():  
    print(key, ":", value)
```

```
a :: 1  
b :: 2  
c :: 3
```

Python 2では、`.keys()` `.values()` および `.items()` は `list` オブジェクトをします。あなたは、にトラフするがあるは、のすることができます `.iterkeys()`、`.itervalues()` と `.iteritems()`

`.keys()` と `.iterkeys()` `.values()` と `.itervalues()` `.items()` と `.iteritems()` ということである `iter*` はジェネレータです。したがって、のは、されるごとに1つずつされます。 `list` オブジェクトがされると、すべてのがリストにバックされ、そのののためにされます。

また、Python 3では、のでされたのはにわないことにもしてください。

Whileループ

`while` ループでは、ループが `false` になるまでループがされます。のコードはループを4します。

```
i = 0  
while i < 4:
```

```
#loop statements
i = i + 1
```

のループは、`for`ループをよりエレガントな`for`ループににできますが、`while`ループは、らかのがたされているかどうかをするのにです。のループは`myObject`がができるまでをけます。

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

`while`ループはなしですることもできますまたはまたは`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             # Prints 1j forever
    print(complex_num)
```

がに`true`の、`while`ループは、`break`ステートメントまたは`return`ステートメントまたはによってされない、にループされます。

```
while True:
    print "Infinite loop"
# Infinite loop
# Infinite loop
# Infinite loop
# ...
```

`pass`は、Python `for`ループまたは`while`ループののなどでステートメントがなときには`null`ステートメントですが、プログラマーはアクションをとしません。これは、まだかれていないコードのブレースホルダとしてちます。

```
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
```

このではもこりません。 `for`ループはエラーなくしますが、コマンドやコードはされません。`pass`は、すべてのコマンドとアクションをにすることなく、コードをにできるようにします。

に、 `while`ループややなどで`pass`をうことができます。

```
while x == y:
    pass
```

なるステップサイズでリストのなるをする

いリストがあり、リストののすべてののにのみがあるとします。おそらくまたはの、またはリストのののエントリだけをべたいとかもしれません。 Pythonはなインデックスをえています。これらのシナリオをするのをいくつかします。

ここでは、をしてされるなリストをします。

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

リストの

リストのをするには、のような `for` ループをできます。

```
for s in lst:
    print s[:1] # print the first letter
```

`for` ループは `lst` に `s` を `for` ます。これはされます

```
a
b
c
d
e
```

くの、とそのインデックスのがです。 `enumerate` キーワードは、そのタスクをします。

```
for idx, s in enumerate(lst):
    print("%s has an index of %d" % (s, idx))
```

インデックス `idx` は0からまり、ごとにインクリメントされ、 `s` はされるをみます。のスニペットはのようにされます

```
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
```

サブリストをする

をりしたいPythonがゼロベースのインデックスをしていることにしてください、 `range` キーワードをしてください。

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

これはします

```
lst at 2 contains charlie
lst at 3 contains delta
```

リストはスライスされてもよい。のスライスは、インデックス1のからステップ2のにします.2つのforループはじforします。

```
for s in lst[1::2]:
    print(s)

for i in range(1, len(lst), 2):
    print(lst[i])
```

のはのようにします

```
bravo
delta
```

インデックスとスライスはのトピックです。

"ハーフループ" do-while

のとはなり、Pythonはdo-untilやdo-whileのをっていませんこれは、がテストされるにコードがされるようにします。しかし、じをするために、 while Trueとbreakをみわせることができます。

```
a = 10
while True:
    a = a-1
    print(a)
    if a<7:
        break
print('Done.')
```

これはされます

```
9
8
7
6
Done.
```

ルーピングとアンパック

たとえば、タプルのリストをループしたい

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

このようなことをするのではなく、

```
for item in collection:
    i1 = item[0]
    i2 = item[1]
    i3 = item[2]
    # logic
```

またはこのようなもの

```
for item in collection:
    i1, i2, i3 = item
    # logic
```

これをにうことができます

```
for i1, i2, i3 in collection:
    # logic
```

これは、タプルだけでなく、ほとんどのタイプのiterableでもします。

オンラインでループをむ <https://riptutorial.com/ja/python/topic/237/ループ>

155: ロギング

Examples

Python ロギングの

このモジュールは、アプリケーションとライブラリのないイベントロギングシステムをするとクラスをします。

ライブラリモジュールによってされるロギングAPIをつなは、すべてのPythonモジュールがロギングにできるため、アプリケーションログにサードパーティのモジュールからのメッセージとされたのメッセージをめることができます。

だから、めることができます

コードの

```
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')
```

```
2016-07-26 18:53:55,332 root          DEBUG    this is a debug test
```

INI ファイルによる

ファイルのがlogging_config.iniであるとしします。ファイルのについては、 [ロギングチュートリアル](#)の [ロギング](#)セクションをしてください。

```
[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

```

に、コードで `logging.config.fileConfig()` をします。

```

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')

```

による

Python 2.7では、のをつをできます。 [PEP 391](#)には、ディクショナリのおよびオプションののリストがまれています。

```

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')

```

のロギング

をするは、 `logging.exception(msg)` メソッドをすることができます。

```

>>> 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
```

としてをさないでください

`logging.exception(msg)` は `msg` `arg` をしているので、のようにロギングびしにをすのはなとしてです

```
>>> 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
```

これがにやるべきことであるかのようにえるかもしれませんが、には、とさまざまなエンコーディングがロギングモジュールでどのようにするのかというがあります。

```
>>> 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
```

ユニコードをむをログにしようとする、このではにします。これは、ののstacktraceを、`logging.exception(e)` びしのフォーマットにしたしいものできすることによってすでしょう。

らかに、あなたのコードでは、のエンコーディングをとっているかもしれません。しかし、サードパーティのライブラリでは、これをのがあります。

しい

のわりに、にメッセージをして、Pythonにそのをさせるなら、それはうまくいくでしょう

```
>>> 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\x66\x66
```

この、には`e`をしないので、`logging.exception(...)`びすと、のがのようにフォーマットされます。

ERRORのログレベルによるのロギング

ERRORよりものログレベルでをログにするは、デフォルトのロガーの`exc_info`をできます。

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

のメッセージへのアクセス

ライブラリーは、ユニコードまたはあなたががければ、utf-8のバイトのいずれかとして、メッセージとともにをスローするがあることにしてください。にのテキストにアクセスするがあるは、`repr(e)`または`%r`のをするだけです。

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('received this exception: %r' % e)
...
ERROR:root:received this exception: Exception(u'f\x66\x66',)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\x66\x66
```

オンラインでロギングをむ <https://riptutorial.com/ja/python/topic/4081/ロギング>

156: ロケールモジュール

Python 2 Docs[<https://docs.python.org/2/library/locale.html#locale.currency>] [1]

Examples

localeモジュールをしたのドル

```
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'
```

オンラインでロケールモジュールをむ <https://riptutorial.com/ja/python/topic/1783/ロケールモジュール>

157: べえ、と

Examples

いくつかののまたはの

```
min(7,2,1,5)
# Output: 1

max(7,2,1,5)
# Output: 7
```

keyの

シーケンスのシーケンスの/をつけることはです

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Output: (0, 10)
```

しかし、シーケンスののでソートするは、 `key -argument`をし `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)]
```

デフォルトの **max**、 **min**

のシーケンスを `max` または `min` すことはできません

```
min([])
```

`ValueError: minarg is the sequence`です

しかし、Python 3では、をさせるわりに、シーケンスがのにされるをキーワード `default` することができます。

```
max([], default=42)
# Output: 42
max([], default=0)
# Output: 0
```

な

またはを `sorted` するかは、オブジェクトのによってなります。 `dict` の、はキーのだけです

```
adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Output: 'a'
max(adict)
# Output: 'c'
sorted(adict)
# Output: ['a', 'b', 'c']
```

をするには、 `.items()` をりしするがあり `.items()` 。

```
min(adict.items())
# Output: ('a', 3)
max(adict.items())
# Output: ('c', 1)
sorted(adict.items())
# Output: [('a', 3), ('b', 5), ('c', 1)]
```

`sorted` は、 `OrderedDict` をして、 `dict` ようなをしながらソートができます。

```
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
```

で

やはりこれは `key` をってです

```
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)]
```

べえられたシーケンスの

1つのシーケンスの

```
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']
```

はにしい `list` です。のデータはされません。

シーケンスのと

シーケンスの `iterable` をすることは、 `sorted` シーケンスののにアクセスすることとじです。

```
min([2, 7, 5])
# Output: 2
sorted([2, 7, 5])[0]
# Output: 2
```

`sorted` はをし、 `max` はにったをすため、はしです。がない、はソートされたリターンのののと同じです。

```
max([2, 7, 5])
# Output: 7
sorted([2, 7, 5])[-1]
# Output: 7
```

しかし、をつとされるのがあるはそうではありません。

```
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
```

＜または＞をサポートするをむなものはすべてされます。

カスタムクラスをにする

`min`、`max`、および `sorted` すべてオブジェクトをにするがあります。にであるためには、クラスが6つのメソッドをすべてするがある `__lt__`、`__gt__`、`__ge__`、`__le__`、`__ne__`と`__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
```

これらのメソッドをすべてするはないようですが、それらのメソッドをするとコードがバグになりやすくなります。

```
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)]
```

reverse=True sortedでも__lt__され__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)]
```

しかしsortedは、デフォルトがされていないはわりに__gt__をできます。

```
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)
```

__lt__も__gt__もされていなければ、ソートメソッドはTypeErrorさせます

```
del IntegerContainer.__gt__ # The IntegerContainer no longer implements "greater then"

res = min(alist)
```

TypeErrorけなIntegerContainer<IntegerContainer

[functools.total_ordering](#)デコレータをすると、これらのなメソッドをにできます。あなたとあなたのクラスをtotal_ordering、あなたがするが__eq__、__ne__との__lt__、__le__、__ge__または__gt__、そしてデコレータはにされます


```

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

```

どのように、よりきいがよりさいメソッドをびすか、によっては__eq__メソッドを__eq__に__eq__。これは、がになは、それぞれのなをでするがあることをします。

iterableからNのアイテムまたはNのアイテムをする

iterableのまたはのいくつかのをつけるには、`heapq`モジュールの`nlargest`と`nsmallest`をします。

```

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]

```

これは、イテラブルをソートし、またはからスライスするよりはるかにです。には、これらのは**バイナリヒープキュー**のデータをします。これは、このユースケースではにです。

`min`、`max`、`sorted`とに、これらのはオプションの`key`キーワードをけります。これは、えられたにしてソートキーをすでなければなりません。

ファイルから1000をするプログラムをにします。

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

ここでファイルをき、ファイルハンドル `f` を `nlargest` ます。ファイルをすると、ファイルのが々のとしてされます。 `nlargest` は `nlargest` をして `len` にし、ソートキーをします。 `len` はでえられ、のさをでします。

これはまでの1000のリストのためのストレージしかとしません。

```
longest_lines = sorted(f, key=len)[1000:]
```

ファイルをメモリにするがあります。

オンラインでべえ、とをむ <https://riptutorial.com/ja/python/topic/252/べえ-と>

158:

GILグローバルインタプリタロックのため、Pythonインタプリタの1つのインスタンスだけが1つのプロセスでされます。に、マルチスレッドをすると、CPUにバインドされたものではなく、IOバウンドのがします。CPUにされたタスクをするは、`multiprocessing`モジュールをおめします。

GILはPythonのもなであるCPythonとPyPyにされます。 [Jython](#)や[IronPython](#)などののでは、GILは [ありません](#)。

Examples

マルチプロセッシング・モジュールをしたタスクの

```
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]
```

`fib` へのびしのがしてわれるので、なのは、デュアルプロセッサでシーケンシャルなでされたよりも**1.8**です。

Python 2.2

スクリプトとスクリプトをしてにコードをする

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()
```

これは、パラレルでしたHTTP/タスクまたはデータベースの/にちます。コマンドラインは **child.py** スクリプトにもえられます。スクリプトのは、すべてのスクリプトがにのサーバーRedis インスタンスなどをチェックすることでできます。

C-エクステンションをしてタスクをする

ここでのえは、のジョブをPythonとはしたCなマクロをにし、CコードをGILがしているにすることです。

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

PyParモジュールをしてする

PyParはPythonでをするためにメッセージパッシングインターフェイスMPIをするライブラリです。PyParのな <https://github.com/daleroberts/pypar>にられるはのようになります

```
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()
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/542/>

159:

き

は、なるプロジェクトでとされるを々のにするためのツールです。をするために、をします。「Project Xはバージョン1.xにしますが、Project Yは4.xがです」というジレンマをし、グローバルなサイトパッケージディレクトリをきれいにします。

これは、おいになるプロジェクトやシステムライブラリからをするのにちます。

は、おそらくすべてのプロジェクトにするがあるほどにです。に、ではのことがです。

1. ルートアクセスをとせずにをする
2. なるのなるバージョンをインストールする。たとえば、さまざまななるプロジェクトでするなど
3. なるPythonバージョンでする

Examples

のと

`virtualenv`は、したPythonをするためのツールです。このプログラムは、Pythonプロジェクトがとするパッケージをするためになすべてのファイルをむフォルダをします。

virtualenv ツールのインストール

これはだけです。あなたのディストリビューションから`virtualenv`プログラムをできます。

Debianのようなディストリビューションでは、パッケージは`python-virtualenv`または`python3-virtualenv`とされます。

わりに、`pip`をして`virtualenv`をインストールすることもできます

```
$ pip install virtualenv
```

しいの

これはプロジェクトごとに1のみです。をりすプロジェクトをするときに、このプロジェクトのしいをできます。

```
$ virtualenv foo
```

これはツールスクリプトと`python`バイナリのコピーを`foo`フォルダをします。フォルダのはありません。がされると、それはであり、`virtualenv`ツールをしてさらにするはありません。のをできます。

のの

をにするには、シェ尔マジックがです。そのため、Pythonはシステムのわりに`foo`にあるものです。これは`activate`ファイルであり、のシェルにソースするがあります

```
$ source foo/bin/activate
```

Windowsユーザーはのようになります。

```
$ foo\Scripts\activate.bat
```

がになると、`python`と`pip`バイナリと、サードパーティのモジュールによってインストールされたすべてのスクリプトが`foo`ます。に、`pip`インストールされたすべてのモジュールは、にされ、がまれています。をアクティブにすると、のコマンドにすように、プロンプトにプレフィックスをするがあります。

```
# Installs 'requests' to foo only, not globally
(foo)$ pip install requests
```

のと

`pip`をしてインストールしたモジュールをするには、`freeze`コマンドをして、これらのモジュールおよびするバージョンをすべてテキストファイルにリストすることができます。これにより、のが、インストールコマンドをって、アプリケーションになPythonモジュールをすばやくインストールすることができます。そのようなファイルののは`requirements.txt`

```
(foo)$ pip freeze > requirements.txt
(foo)$ pip install -r requirements.txt
```

`freeze`は、でインストールしたトップレベルモジュールがとするなをめ、すべてのモジュールをリストしています。したがって、なトップレベルのモジュールだけをくことによって、`requirements.txt`ファイルをでることをむかもしれません。

の

でのがわったら、それをにしてのシェルにすことができます

```
(foo)$ deactivate
```

ホストでの

ホストでmod_wsgiをしているや、Amazon API GatewayやGoogle AppEngineなどのファイルシステムにアクセスできないなど、`$ source bin/activate INBIT $ source bin/activate virtualenv`を`$ source bin/activate`できないことがあります。そのようなのために、あなたはあなたの`virtualenv`のにインストールされたライブラリをし、あなたのパッチをすることができます
`sys.path`。

あなたののスク립トと Luckily `virtualenv`の`sys.path`とあなた`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))
```

これらは、サーバーがするファイルのにするがあります。

これは`bin/activate_this.py` `virtualenv`があなたがしているじディレクトリにファイルをし、`lib/python2.7/site-packages`を`sys.path`します

`activate_this.py` スクリプトをするは、`bin`と`lib/python2.7/site-packages`ディレクトリとそのでなくともしてください。

Python 3.x 3.3

ビルト イン

Python 3.3では、`venv`モジュールはをします。 `pyvenv` コマンドはにインストールするはありません。

```
$ pyvenv foo
$ source foo/bin/activate
```

または

```
$ python3 -m venv foo
$ source foo/bin/activate
```

へのパッケージのインストール

がアクティブになると、インストールするパッケージは`virtualenv`インストールされ、グローバルにはインストールされません。したがって、しいパッケージは`root`をとせずにみます。

パッケージが`virtualenv`にインストールされていることをするには、の`コマンド`をして、されてい

るファイルのパスをします。

```
(<Virtualenv Name>) $ which python
/<Virtualenv Directory>/bin/python

(Virtualenv Name) $ which pip
/<Virtualenv Directory>/bin/pip
```

pipをしてインストールされたパッケージは、 virtualenv のディレクトリにインストールされます。

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

あるいは、なパッケージをリストしたファイルをすることもできます。

requirements.txt

```
requests==2.10.0
```

```
# Install packages from requirements.txt
pip install -r requirements.txt
```

バージョン2.10.0のパッケージ requests がインストールされ requests 。

アクティブなにインストールされているパッケージとそのバージョンのリストをすることもできます。

```
# 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
```

また、パッケージをインストールするたびにをアクティブするはありません。パッケージをインストールするには、ディレクトリのpipファイルをすることができます。

```
$ /<Virtualenv Directory>/bin/pip install requests
```

pipのにするは、 [PIPのトピックをしてください](#) 。

にルートなしでインストールしているので、これはシステムにわたるグローバルインストールではなく、インストールされたパッケージはのでのみになります。

なるバージョンのPythonのをする

するpythonとpython3インストールされている、でも、Pythonの3のためのをすることがであるpython3デフォルトのPythonではありません。

```
virtualenv -p python3 foo
```

または

```
virtualenv --python=python3 foo
```

または

```
python3 -m venv foo
```

または

```
pyvenv foo
```

には、システムのPythonのバージョンについてをすることができます。あなたは、あなたの
のpythonをすることができます /usr/bin/または/usr/local/bin/ LinuxのORで
/Library/Frameworks/Python.framework/Versions/XX/bin/、その、OSXをつけしますをする--python
に、 - --pythonまたは-pフラグでそのをします。

virtualenvwrapperをってのをする

[virtualenvwrapper](#)ユーティリティは、でのをし、くのプロジェクトをうににです。

[virtualenvwrapper](#)は、ディレクトリをで[virtualenvwrapper](#)するわりに、すべてのをディレクトリデ
フォルトで ~/.virtualenvs にすることで、それらをします。

インストール

システムのパッケージマネージャーに[virtualenvwrapper](#)をインストールします。

Debian / Ubuntuベース

```
apt-get install virtualenvwrapper
```

Fedora / CentOS / RHEL

```
yum install python-virtualenvwrapper
```

アーチLinux

```
pacman -S python-virtualenvwrapper
```

またはしては、PyPIからインストール `pip`

```
pip install virtualenvwrapper
```

Windowsでは、わりに`virtualenvwrapper-win`または`virtualenvwrapper-powershell`をできます。

は`mkvirtualenv`され`mkvirtualenv`。 の`virtualenv`コマンドのもすべてけられます。

```
mkvirtualenv my-project
```

または

```
mkvirtualenv --system-site-packages my-project
```

しいがにアクティブになります。しいシェルでは、`workon`をにすることができます

```
workon my-project
```

のものとして、`workon`コマンドの`. path/to/my-env/bin/activate`は、`workon`コマンドがどのディレクトリでもすることをしています。プロジェクトののがされているディレクトリをえておくはありません。

プロジェクトディレクトリ

`setvirtualenvproject`コマンドをして`-a`オプションでをするときプロジェクトディレクトリをすることもできます。

```
mkvirtualenv -a /path/to/my-project my-project
```

または

```
workon my-project
cd /path/to/my-project
setvirtualenvproject
```

プロジェクトをすると、`workon`コマンドがプロジェクトににりえられ、プロジェクトディレクトリにできる`cdproject`コマンドがになります。

`virtualenvwrapper`によってされるすべての`virtualenv`のリストをするには、`lsvirtualenv`をし`lsvirtualenv`。

`virtualenv`をするには、`rmvirtualenv`をし`rmvirtualenv`。

```
rmvirtualenv my-project
```

`virtualenvwrapper`によってされている`virtualenv`には、`preactivate`、`postactivate`、`predeactivate`、`preactivate`、および`postdeactivate` `predeactivate` `preactivate` 4つの`bash`スクリプトがまれています。これらは、`virtualenv`のライフサイクルのあるで`bash`コマンドをするためのフックとしてします。たとえば、`postactivate`スクリプトのコマンドは、`virtualenv`がアクティブになったにされ

ます。これはな、エイリアス、またはするものをするのにしています。4つのスクリプトはすべて、`.virtualenvs/<virtualenv_name>/bin/`ます。

は、[virtualenvwrapperのドキュメント](#)をしてください。

しているの

Linuxでデフォルトの`bash`プロンプトをしているは、プロンプトののにのがされます。

```
(my-project-env) user@hostname:~$ which python
/home/user/my-project-env/bin/python
```

Unix / LinuxのスクリプトでするのPythonバージョンをする

LinuxシェルがうべきPythonのバージョンをするために、Pythonスクリプトののは、`#!`でまるシバンにすることができます`#!`。

```
#!/usr/bin/python
```

あなたは、にある、`python myscript.py`からのPythonをしますが、`./myscript.py`でPythonインタプリタをします`#!`ライン。のPythonがされていることをするには、のをのようになります。

```
#!/usr/bin/env python
```

シバンをしたら、のようにしてスクリプトにをえてください。

```
chmod +x myscript.py
```

これをうと、あなたがしてスクリプトをできるようになります。`./myscript.py`のわりにまたはスクリプトへのパスをする`python myscript.py`か`python3 myscript.py`。

ので**virtualenv**をう

のシェルはしみやすいですが、`virtualenv`または`virtualenvwrapper`をしている`virtualenv`にするがあります。あるいは、のために`virtualfish`がします。Fishenシェルを`virtualenv`でいめるには、のにってください。

- グローバルスペースに`virtualfish`をインストールする

```
sudo pip install virtualfish
```

- のシェルのに、Pythonモジュールのバーチャルフィッシュをみむ

```
$ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
```

- `$ funced fish_prompt --editor vim`

この`fish_prompt`する - `vim`をし、のをして`vim`エディタをじる

```
if set -q VIRTUAL_ENV
    echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color
normal) " "
end
```

`vim`にれていないは、このようなみのエディタを`$ funced fish_prompt --editor nano`または`$ funced fish_prompt --editor gedit`

- `funcsave`をつてをする

```
funcsave fish_prompt
```

- しいをするには、`vf new`します。

```
vf new my_new_env # Make sure $HOME/.virtualenv exists
```

- しい`python3`をしたいは、`-p`フラグでします

```
vf new -p python3 my_new_env
```

- をりえるには、`vf deactivate` `vf activate` `another_env`します。

リンク

- <https://github.com/adambrenecki/virtuallfish>
- <http://virtuallfish.readthedocs.io/en/latest/>

Anacondaをしたの

`virtualenv`のな`virtualenv`は、をくおよびするためのをバンドルした、クロスプラットフォームの`pip - virtualenv`パッケージマネージャーである**Anaconda**です。 **Anaconda**をインストールした、するコマンドがいくつかあります

をする

```
conda create --name <envname> python=<version>
```

<envname>はなので、 <version>はするのPythonバージョンです。

のアクティブとアクティブ

```
# Linux, Mac
source activate <envname>
source deactivate
```

または

```
# Windows
activate <envname>
deactivate
```

されたのリストをする

```
conda env list
```

をする

```
conda env remove -n <envname>
```

よりくのコマンドとについては、の[ドキュメント](#)をしてください。

でされているかどうかの

によっては、シェルプロンプトにのがされず、にいるかどうかをしたいことがあります。

Pythonインタプリタをしてしてみてください

```
import sys
sys.prefix
sys.real_prefix
```

- のでは、 `sys.prefix`はシステムPythonのインストールを `sys.real_prefix`おり、
`sys.real_prefix`はされていません。
- ので、 `sys.prefix`はpythonのインストールを `sys.real_prefix`、 `sys.real_prefix`はシステムの
pythonのインストールをします。

ライブラリ[venvモジュール](#)をしてされたでは、 `sys.real_prefix`はありませ `sys.real_prefix`。 わりに、かどうかを `sys.base_prefix`じである `sys.prefix`。

オンラインでをむ <https://riptutorial.com/ja/python/topic/868/>

160:

き

にされたエラーはとばれ、にではありません。ほとんどののはプログラムによってされません。したをするプログラムをすることができます。Pythonには、やロジックをうのがあります。さらに、にはながあり、すべて `BaseException` をしています。

- をする
- `raise`すでにしているをさせる
- を#Pythonからきげる3 - のをする
- をNoneからすPython 3 - すべてのコンテキストをする
- してください
- [として][タイプ]をきます。
- `else`
- に

Examples

の

コードがにした、なパラメータなどのがわからないは、ながするはずです。

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an odd number")

    return odds + 1
```

のキャッチ

をキャッチするには、`try...except:`をします。のように、なをするがあります。

```
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.
```

されたクラスこのは `ZeroDivisionError` は、そのクラスまたはそののサブクラスであるすべてのをキャッチします。

たとえば、`ZeroDivisionError`は`ArithmeticError`サブクラスです。

```
>>> ZeroDivisionError.__bases__  
(<class 'ArithmeticError'>,,)
```

したがって、の`ZeroDivisionError`ます

```
try:  
    5 / 0  
except ArithmeticError:  
    print("Got arithmetic error")
```

にクリーンアップコードをする

によっては、らかのリソースがクリーンアップされなければならないなど、がしたでも、するものがあります。

`try`の`finally`ブロックは、がしたかどうかにかかわらずします。

```
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)
```

このパターンは、`with`ステートメントをし`with` コンテキストマネージャーでよくわれることがよくあります。

の

によっては、えばログをするために、するためにをキャッチすることがあります。の、とじようにがしけるようにします。

この、なしで`raise`をするだけです。

```
try:  
    5 / 0  
except ZeroDivisionError:  
    print("Got an error")  
    raise
```

ただし、びしスタックののユーザーがをキャッチしてとかできることにしてください。したは、いずれのにもするため、このはになるがありますキャッチまたはキャッチされません。だから、のとにについてのあなたのコメントをむのをさせるがいかもしれません

```
try:
```



```
5 / 0
except ZeroDivisionError as e:
    raise ZeroDivisionError("Got an error", e)
```

しかしこれには、トレースをにこの`raise`にらすというがあり、なしの`raise`はのトレースをします。

Python 3では、`raise - from`をつてのスタックをすることができます

```
raise ZeroDivisionError("Got an error") from e
```

のチェーン

をするプロセスでは、のをさせることができます。たとえば、ファイルからみみに`IOError`がした、アプリケーションのエラーをさせて、ライブラリのユーザーにすることができます。

Python 3.x 3.0

をどのようにしたかをすために、をすることができます。

```
>>> 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
```

は、クラスのによってされるについてわれます。

たとえば、`IOError`と`OSError`はとも`EnvironmentError`サブクラスです。`IOError`をする`OSError`は`OSError`しません。ただし、`EnvironmentError`をするコードは、`IOError`と`OSError`のをし`IOError`。

みみの

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

```

もオブジェクトです

は、みみの `BaseException` をするなるの Python オブジェクトです。Python スクリプトは `raise` をしてをすることができ、Python はそのでびしスタックのスタックトレースとインスタンスのをします。えは

```

>>> 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!
```

'Example error!'というメッセージをむ`ValueError`れ、'Example error!'たちの`failing_function()`によって`failing_function()`れました。これはインタプリタでされました。

コーリングコードは、コールがさせることができるすべてのタイプののをすることができます。

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Handled the error')
Handled the error
```

コードの`except...`にオブジェクトをりてることによって、オブジェクトをすることができます。

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Caught exception', repr(e))
Caught exception ValueError('Example error!')
```

みみのPythonのなリストとそのは、Pythonドキュメント

<https://docs.python.org/3.5/library/exceptions.html>にあります。そして、ここではでされたなリストがあります [Exception Hierarchy](#)。

カスタムのをする

Exceptionからするクラスをする

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
except FooException:
    print("A FooException was raised.")
```

またはのタイプ

```
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))
```

すべてをキャッチしないでください

それはしばしばすべての `Exception` をキャッチするために `Exception` ですが、

```
try:
    very_difficult_function()
except Exception:
    # log / try to reconnect / exit gracefully
finally:
    print "The END"
    # it runs no matter what execute.
```

あるいは、すべて `BaseException` と `Exception` をむすべてのをむ

```
try:
    even_more_difficult_function()
except:
    pass # do whatever needed
```

ほとんどの、それはいいです。 `SystemExit`、`KeyboardInterrupt`、`MemoryError` など、したのものをまえるがあります。は、のシステムエラーやロジックエラーとはなるがわれます。また、コードがっているがあり、そのからにするについてながないことをします。あなたがすべてのエラーをキャッチしているは、どのようなエラーがしたのか、それをするはわかりません。

これはよりに「バグマスキング」とばれ、けるべきです。ってするか、するのではなく、プログラムをクラッシュさせ、よりいレベルでするようにしてください。それがトランザクションシステムであるとしてください

、これらののはプログラムののにのレベルでされ、バグをできるようにエラーのをするか、エラーをよりにうことができます。

のをキャッチする

のをキャッチするはいくつかあります。

1つは、じでキャッチしてするタイプのタプルをすることです。このでは、コードで `KeyError` と `AttributeError` がされます。

```
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.")
```

なるをさまざまでしたいは、タイプごとに々のブロックをすることができます。このでは、きき`KeyError`と`AttributeError`キャッチし`KeyError`が、をさまざまでします。

```
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)
```

の

ユーザー

ユーザーがをしてを`input`。がであることをするがあります。 `try / except`をすることができます

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.')
```

Python 2.xはわりに`raw_input`をします。 `input`はPython 2.xではし`input`が、セマンティクスはなります。のでは、 `input`は`2 + 2`ようなをけり、にされます。

をにできない`ValueError`は、 `ValueError`がします。あなたはでそれをキャッチすることができます `except`。 がしなければ、 `break`はループからびします。ループの、 `nb`はをむ。

`range(n)`ようなしたのリストをしているとし、のにしたときにうべきことにするがっている`d`リストをとっているとしたら、 `d[i]`スキップします。

```
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
```

しないキーのからをしようとする、 `KeyError`がします。

その

`else` ブロックのコードは、`try` ブロックのコードによってがしていないにのみされます。これは、`try` ブロックのコードによってスローされたコードがいくつかあるが、そのコードによってスローされたをまえないにです。

例えば

```
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# Output: one
# Output: ValueError
```

この `else: elif` `else` をする `if` は、とみわせることはできません。にインデントされているがある `if` は、の `if` で `else:`

```
try:
    ...
except ...:
    ...
else:
    if ...:
        ...
    elif ...:
        ...
    else:
        ...
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/1788/>

161:

き

は、テキストのをのテキストにします。これらのは、PythonでOCRをするをつける。

Examples

PyTesseract

PyTesseractは、OCRののPythonパッケージです。

PyTesseractをするのはです

```
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はオープンソースであり、[ここでつけることができます](#)。

PyOCR

いくつかののもうつのモジュールは、PyOCR、そのソースコードは[ここにあります](#)。

PyTesseractよりもPyTesseractやすく、もPyTesseractです。

するには

```
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.
```

そしていくつかの

```
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
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/9302/>

162: でないPythonの

Examples

IronPython

Cでかかれた.NETとMonoのオープンソースで、Apache License 2.0でライセンスされています。DLRランタイムにしています。これはバージョン2.7のみをサポートし、バージョン3はです。

CPythonとのい

- .NET Frameworkとのな。
- はデフォルトでUnicodeです。
- CでかかれたCPythonのをサポートしていません。
- グローバルインタープリタロックにまされません。
- パフォーマンスはテストにしますが、はパフォーマンスはします。

こんにちは

```
print "Hello World!"
```

.NETをすることもできます。

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

リンク

- [サイト](#)
- [GitHubリポジトリ](#)

Jython

JavaでかかれたJVMのオープンソースで、Python Software Foundation Licenseでライセンスされています。これはバージョン2.7のみをサポートし、バージョン3はです。

CPythonとのい

- JVMとのな。
- はUnicodeです。

- CでかかれたCPythonのをサポートしていません。
- グローバルインタープリタロックにまされません。
- パフォーマンスはテストにしますが、パフォーマンスはします。

こんにちは

```
print "Hello World!"
```

Javaをすることもできます。

```
from java.lang import System
System.out.println("Hello World!")
```

リンク

- [サイト](#)
- [Mercurialリポジトリ](#)

トランスクリプト

Transcriptは、PythonのなをコンパクトでみやすいJavaScriptにプリコンパイルするためのツールです。それにはのがあります

- CPythonのネイティブパーサーによってされたなPythonをして、のをつなOOプログラミングをにする
- デスクトップのPythonスクリプトではなく、のウェブのJavaScriptライブラリとのシームレスな
- PyPiのモジュールをにするなURLベースのモジュールシステム
- なデバッグのためのPythonソースとされたJavaScriptコードとのな
- マルチレベルのソースマップと、ソースコードをむターゲットコードのオプションのアノテーション
- コンパクトなダウンロード、MBのではなくkB
- オプションでプロトタイプのルックアップチェーンをバイパスするためのmemoizationコールキャッシングをしたされたJavaScriptコード
- のオーバーロードをローカルでオン/オフにりえることで、がになります。

コードサイズと

によると、650 kBのPythonソースコードは、じのJavaScriptソースコードでほぼされています。はきJavaScriptのとし、メモがオンになっているはそれをるがあります。

HTML との

```
<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>
```

JavaScript と DOM との

```
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 ()
```

の JavaScript ライブラリ との

Transcriptは、なやをしなくても、のJavaScriptライブラリとみわせてできます。ドキュメントでは、ao react.js、riot.js、fabric.js、node.jsのをしています。

PythonとJavaScriptコードの

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');
```

リンク

- サイト [http : //www.transcrypt.org/](http://www.transcrypt.org/)
- リポジトリ [https : //github.com/JdeH/Transcrypt](https://github.com/JdeH/Transcrypt)

オンラインででないPythonのをむ <https://riptutorial.com/ja/python/topic/5225/でないpythonの>

163: とし

き

Pythonはあいまいさやしないるいがなく、でみやすいです。ながら、これらはすべてののにはありません。そのため、Pythonはあなたがしていたものとはなるかをうかもしれないいくつかのコーナーケースをとっています。

このセクションでは、Pythonコードをくときにするがあるいくつかのをします。

Examples

のシーケンスをする

forループはシーケンスをするので、ループでこのシーケンスをすると、にエレメントのやにしないになります。

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]
```

list.pop()は、リストからをするためにされています。

2のは、イテレーションがにインデックスをするため、されませんでした。のループは2され、のがされます。

```
# 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]
```

このは、インデックスがするにしながらインデックスがしているためにします。このをするには、ループをにすることができます。

```
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]
```

からループをすることで、がまたはされるので、リストのののインデックスにはしません。したがって、このでは、`alist` からすべてのアイテムがにされ`alist`。

のリストにをまたはすると、のがし、ループがするがあります。

```
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]
```

`break`がなければ、コンピュータはメモリでプログラムがされないうり、ループは、`'a'`をします。このようなでは、しいリストをし、のリストをループしながらしいリストにをすることをおめします。

`for`ループをするは、プレースホルダをしてリストをすることはできません。

```
alist = [1,2,3,4]
for item in alist:
    if item % 2 == 0:
        item = 'even'
print(alist)
# Out: [1,2,3,4]
```

のでは、`item`しものリストではもされません。リストインデックス `alist[2]` をするがあり、`enumerate()` はこれにしてうまくいきます

```
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']
```

いくつかのケースでは`while`ループをするがいでしょう。

リストのすべてのをするは、のようにします。

```
zlist = [0, 1, 2]
while zlist:
    print(zlist[0])
    zlist.pop(0)
```



```
print('After: zlist =', zlist)

# Out: 0
#      1
#      2
# After: zlist = []
```

に `zlist` をリセットするとじがられますが、

```
zlist = []
```

のは `len()` とみわけて、あるのですること、リストの `x` ののすべてをすることもできます

```
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]
```

または、のをたすをしながらリストをループすることができます この、すべてのをします。

```
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]
```

をしたは、`i` をインクリメントしないことにしてください。あるをすることによって `zlist[i]` チェックすることによって、ののインデックスは、1している `zlist[i]` ため、じをつ `i` のので、あなたはしく、リストののをチェックします。

なアイテムをリストからすることをえると、のは、しいリストになアイテムをすることです。のでは、ので `while` ループの

```
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]
```

ここでは、ましいをしいリストにめています。その、オプションでリストをのにりてすることができます。

このようなので、Pythonのもっともエレガントでな、 リストをびすことで、なリストをし、のインプレースリスト/インデックスのイデオロギーからすることができます。

```
zlist = [1,2,3,4,5]
[item for item in zlist if item % 2 != 0]
# Out: [1, 3, 5]
```

なデフォルト

```
def foo(li=[]):
    li.append(1)
    print(li)

foo([2])
# Out: [2, 1]
foo([3])
# Out: [3, 1]
```

このコードはりにしますが、をさないとうなりますか

```
foo()
# Out: [1] As expected...

foo()
# Out: [1, 1] Not as expected...
```

これは、とメソッドのデフォルトのが、ではなくにされるためです。ですから、たちは`li` リストのインスタンスを1つしかてません。

それをするは、デフォルトののにのだけをうことです

```
def foo(li=None):
    if not li:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]

foo()
# Out: [1]
```

されていますが`if not li`が`False`にしくされても、さゼロのシーケンスなど、くのオブジェクトもにします。ののは、しないをきこすがあります。

```
x = []
foo(li=x)
# Out: [1]
```

```
foo(li="")
# Out: [1]

foo(li=0)
# Out: [1]
```

なアプローチは、`None`オブジェクトにしてをチェックすることです。

```
def foo(li=None):
    if li is None:
        li = []
    li.append(1)
    print(li)

foo()
# Out: [1]
```

とのリスト

のようにけわせてれのリストをするをえてみましょう。

```
li = [[]] * 3
print(li)
# Out: [[], [], []]
```

すると、3つのなるれリストをむリストがあるといます。のものに`1`をしようとしましょう

```
li[0].append(1)
print(li)
# Out: [[1], [], []]
```

`1`が`li`すべてのリストにされました。

そのは、`[[]] * 3`は3つのなる`list`の`list`をしないからです。むしろ、じ`list`オブジェクトへの3つのをする`list`をします。したがって、`li[0]`すると、そのは`li`すべてのでされます。これはとじです

```
li = []
element = [[]]
li = element + element + element
print(li)
# Out: [[], [], []]
element.append(1)
print(li)
# Out: [[1], [1], [1]]
```

これは、`id`をつて、まれている`list`メモリアドレスをすると、さらにけられます

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# Out: [6830760, 6830760, 6830760]
```

は、リストをループですることです。

```
li = [[] for _ in range(3)]
```

1つの`list`をして3つのをするのではなく、3つのなるリストをします。これは、`id`をしてすることが出来ます。

```
print([id(inner_list) for inner_list in li])  
# Out: [6331048, 6331528, 6331488]
```

あなたもこれをうことができます。これにより、`append`びしでしいのリストがされ`append`。

```
>>> li = []  
>>> li.append([])  
>>> li.append([])  
>>> li.append([])  
>>> for k in li: print(id(k))  
...  
4315469256  
4315564552  
4315564808
```

シーケンスをループするためにインデックスをししないでください。

しないでください

```
for i in range(len(tab)):  
    print(tab[i])
```

う

```
for elem in tab:  
    print(elem)
```

`for` あなたのためのものもをします。

インデックスとののがになは、をします。

```
for i, elem in enumerate(tab):  
    print((i, elem))
```

"=="をして**True**または**False**をチェックするはしてください

```
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

```

できるかどうかはしないでください。エラーをしてください

Pythonistasは、「よりもしをめるのはです」といいます。

しないでください

```

if os.path.isfile(file_path):
    file = open(file_path)
else:
    # do something

```

う

```

try:
    file = open(file_path)
except OSError as e:
    # do something

```

Python 2.6+さらにれていPython 2.6+

```

with open(file_path) as file:

```

はるかになので、はるかにれています。ほとんどのものにtry/exceptをすることができます。あなたはそれをぐためにをすべきかするはありません。あなたがにさらされているエラーにをけるだけです。

タイプにしてチェックしない

Pythonはにけされているため、のチェックでがわれます。わりに、をしてダックタイピングをしてください。にがあるとされるは、str()をしてのオブジェクトをにします。リストがなは、list list()をしてのイテレートを一リストにします。

しないでください

```
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)
```

う

```
def foo(name) :
    print(str(name).lower())

def bar(listing) :
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

のをして、`foo`はのオブジェクトをくれます。 `bar`は、、タプル、セット、リストなどをくれます。**DRY**。

スペースとタブをさせないでください

オブジェクトをのとしてする

これはやりにくいですが、プログラムがするにつれてあなたをんでしまいます。 `Python 2.x`はいクラスとしいクラスがあります。いものは、よく、いものです。らはいくつかのをいており、でいにくいことがあります。できるようにするには、クラスのいずれかが「しいスタイル」でなければなりません。これをうには、`object`からさせ `object`。

しないでください

```
class Father:
    pass

class Child(Father):
    pass
```

う

```
class Father(object):
    pass

class Child(Father):
    pass
```

`Python 3.x`すべてのクラスがしいスタイルなので、そうするはありません。

initメソッドのでクラスをしない

のからているは、それがJavaやPHPでやっているのです。クラスをし、をリストしてデフォルトをえます。しかし、Pythonではうまくいくようですが、これはあなたがえるようにはしません。そうすることで、クラスがされ、オブジェクトをしようとするのでないりがられます。その、クラスをします。それは2つのきなをします

- classがされた、がされます。
- オブジェクトをデフォルトとしてすると、じオブジェクトがインスタンスでされます。

しないでくださいにしないり

```
class Car(object):
    color = "red"
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

う

```
class Car(object):
    def __init__(self):
        self.color = "red"
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

との

Pythonは、のにしてキャッシュをして、りしのからなオーバーヘッドをらしします。

には、これはアイデンティティをするときにするにつながるがあります。

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
True
```

のをすると、

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

って

たちは、アイデンティティのがあることがわかりますisりTrueいくつかのの-3、256のためが、-8、257。

には、[-5, 256]のはインタプリタののにキャッシュされ、だけされます。このように、これらはであり、でのアイデンティティをすることisりTrue。こののはオンザフライでされ、そのIDはFalseとされます。

これはテストのなであるため、なとしです。しかし、くの、コードはのステージングプロセスまたはいでします。

は、アイデンティティ `is` で `is` なく、 `==` をしてにをすることです。

Pythonはにされるへのをし、のアイデンティティつまり `is` をするをするときののをく `is` あります。

```
>>> 'python' is 'py' + 'thon'
True
```

'python'というがよくわれるので、Pythonは'python'というへのがすべて1つのオブジェクトをしています。

でないの、がしいでものにします。

```
>>> '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
```

したがって、Integersのルールとじように、にアイデンティティ `is` で `is` なく `==` をしてをします。

int リテラルへのアクセス

Pythonのすべてがリテラルであっても、オブジェクトであるといったことがあります。これは、たとえば、`7`もオブジェクトであることをします。これはをつことをします。たとえば、これらのの1つは`bit_length`です。びされたをすためになビットをします。

```
x = 7
x.bit_length()
# Out: 3
```

のコードがをて、あなたはだとうかもしれません`7.bit_length()`のみそれがつけるために、にする `SyntaxError`。 どうしてなぜならインタープリタはアクセスとえば`7.2`または`7.bit_length()` をするがあるからです。それはできないので、がします。

`int` リテラルのにアクセスするには、いくつかのがあります。

```
# parenthesis
(7).bit_length()
# a space
7 .bit_length()
```

この、2つのドットこの`7..bit_length()` をすると、 `float` リテラルがされ、 `float` は`bit_length()` メソッドがないため、このはしません。

これは、`float` リテラルに2つをめることができないことをするのに "スマート"なので、`float` リテラルのにアクセスするときにはしません、例えば

```
7.2.as_integer_ratio()
# Out: (8106479329266893, 1125899906842624)
```

チェインまたはオペレータ

いくつかののいずれかをテストするとき

```
if a == 3 or b == 3 or c == 3:
```

これをするとです

```
if a or b or c == 3: # Wrong
```

これはっています;`or`オペレータがするより`==`のようにされるように、`if (a) or (b) or (c == 3):`しいは、すべてののをにチェックすることです。

```
if a == 3 or b == 3 or c == 3: # Right Way
```

わりに、みみの`any()`を`or`のわりにすることもできます。

```
if any([a == 3, b == 3, c == 3]): # Right
```

あるいは、よりにするために

```
if any(x == 3 for x in (a, b, c)): # Right
```

または、それをくする

```
if 3 in (a, b, c): # Right
```

ここでは、`in`をして`in`するをむタプルにがするかどうかをテストします。

に、

```
if a == 1 or 2 or 3:
```

これは

```
if a in (1, 2, 3):
```

`sys.argv [0]`はされているファイルのです

`sys.argv[0]` ののは、されているPythonファイルのです。りのはスクリプトです。

```
# 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']
```

はけられていません

Pythonは、ええC++ `std::map` ようなキーでソートされるとされるかもしれませんが、そうではありません

```
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には、キーでをにソートするビルトインクラスはありません。

ただし、ソートがではなく、キー/ペアのをにさせたいは、`collections.OrderedDict` をできます。

```
from collections import OrderedDict

oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])

print([k for k in oDict])
# Out: ['first', 'second', 'third']
```

で`OrderedDict` をしても、`OrderedDict` はしてべえられません。このがすることは、キーのをすることだけです。

のはPython 3.6でメモリをするためにされました。このしいのは、にされるキーワードのもするということです。

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
```

ことをしてください「このしいのためのは、のとみなされ、すべきではない」それはされるとき、。

グローバルインタープリタロックGILとスレッドのブロック

PythonのGILについてたくさんかれています。マルチスレッドマルチプロセスとしないでくださいアプリケーションをうときにをくことがあります。

ここにがあります

```
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")
```

スレッドがされたにCalculating...されることがされますが、のところがしいスレッドでがされたかったのですしかし、には、がしたにされることがわかります。これは、しいスレッドがにGILをロックするC `math.factorial` にしているためです。

これにはいくつかのがあります。は、ネイティブPythonでをすることです。これによりループにメインスレッドがをつことができます。は、Cをもうしていないので、このはかなりなることです。

```
def calc_fact(num):
    """ A slow version of factorial in native Python """
    res = 1
    while num >= 1:
        res = res * num
        num -= 1
    return res
```

をにするにsleepすることもできます。これはにあなたのプログラムがCでこっているををするのをしませんが、あなたがしているかもしれないあなたのメインスレッドがスプーンのにくことをします。

```
def calc_fact(num):
    sleep(0.001)
```

```
math.factorial(num)
```

リストのループののれ

のリストのをえてみましょう

Python 2.x 2.7

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 2
```

これは、リストのがループをのスコープ [ソース](#) に「らす」というのために、Python 2でのみします。このは、つけにくいバグにつながり、Python 3でされています。

Python 3.x 3.0

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 0
```

に、forループはのプライベートスコープをたない

```
i = 0
for i in range(3):
    pass
print(i) # Outputs 2
```

このタイプのは、Python 2とPython 3のでします。

リークするのをするには、リストおよびしいループをなでします。

xyzは、aとbの2つのをします。

```
def xyz():
    return a, b
```

コードをびすxyzは、xyzが1つのしかさないとして、を1つのにします。

```
t = xyz()
```

`t`、のタプルa、bはこれにのアクションであり`t`がタプルがタプルについてしないエラーをするコードにいまするがされていないとします。

`TypeError: tupleは...メソッドをしていません`

はのようになります

```
a, b = xyz()
```

はタプルのエラーメッセージをむだけでこのメッセージのをつけるのがいいでしょう

PythonのJSONキー

```
my_var = 'bla';  
api_key = 'key';  
...lots of code here...  
params = {"language": "en", my_var: api_key}
```

JavaScriptにれている、Pythonのはりではありません。 JavaScriptのこのステートメントは、の
ように`params`オブジェクトになります。

```
{  
  "language": "en",  
  "my_var": "key"  
}
```

ただし、Pythonではのようになります。

```
{  
  "language": "en",  
  "bla": "key"  
}
```

`my_var`がされ、そのがキーとしてされます。

オンラインでとしをむ <https://riptutorial.com/ja/python/topic/3553/>とし

164:

をするには、に `stop condition` `stopCondition` がです。

のは、にされてされるがあります。

Examples

1からnまでのの

n である 1 から n までののをべるには、 $1 + 2 + 3 + 4 + \dots + (\text{several hours later}) + n$ ます。わり、は `for` ループをくことができます

```
n = 0
for i in range (1, n+1):
    n += i
```

あるいは、とばれるをすることもできます。

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

にはの2つのよりもがあります。は1から3のにして $1 + 2 + 3$ をきすよりもがかかりません。
`recursion(4)`、をしてにすることができます。

びし $4 \rightarrow 4 + 3 \rightarrow 4 + 3 + 2 \rightarrow 4 + 3 + 2 + 1 \rightarrow 10$

`for` ループはに `forward` $1 \rightarrow 1 + 2 \rightarrow 1 + 2 + 3 \rightarrow 1 + 2 + 3 + 4 \rightarrow 10$ でします。には、はよりです。これは、リンクされたリストのをするときにはらかです。

の、、および

は、のびしがするにびしによってじがびされたにします。えは、よくられている $x!$ すなわち。は、すべてののにしてのようになれます。

- が0の、えは1です。
- それの、えはそのとそのより1ののです。

Pythonでは、のなはのようなとしてできます。

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

は々するのがしいかもしれないので、このステップをみましょう。 `factorial(3)` えてみましょう。このびしとすべてのびしは、しいをします。はに、えは、 `n`、 `factorial`、 `print` などをするにマップするなるテーブルです。どのでも、 `locals()` `locals()` をしてのにアクセスできます。のびしでは、されるのローカルは `n = 3` です。したがって、 `locals()` は `{'n': 3}` ます。 `n == 3`、りは `n * factorial(n - 1)` ます。

こののステップでは、がしするかもしれません。たちのしいをて、はすでに `n` がであるかをついています。しかし、たちはどの `factorial(n - 1)` あるかはまだかりません。まず、 `n - 1` は `2` されます。に、 `2` を `n` のとして `factorial` す。これはしいびしであるため、このしい `n` をするための `2` のがされます。 `A` を `1` の、 `B` を `2` のとする。 `A` はまだし、 `{'n': 3}` にしくなりますが、 `B` `{'n': 2}` しくなりますはのです。をと、りは `n * factorial(n - 1)` です。このをせず、のりにしましょう。これをうことで、たちはに `B` をしているので、それにじて `n` をすることをれないでくださいつまり、 `B` の `n` へのは `A` の `n` をする `n - 1` きえられます。さて、のりは、 `n * ((n - 1) * factorial((n - 1) - 1))` ます。これがなぜそうであるかをするために、しをかけてください。

さて、その `factorial((n - 1) - 1)` をしましょう。 `A` の `n == 3`、 `1` を `factorial` します。したがって、 `{'n': 1}` しいしい `C` をしています。ここでも、りは `n * factorial(n - 1)` です。のりののをのうにしたのとじように、"の"りの `factorial((n - 1) - 1)` をきえましょう。「の」は、 `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))` です。

ほぼしました。、 `factorial((n - 2) - 1)` をするがあります。は `0` をしてい `0`。したがって、これは `1` されます。さて、のをしましょう。「の」りは、 `n * ((n - 1) * ((n - 2) * 1))` です。のりが `A` でされることをいすと、は `3 * ((3 - 1) * ((3 - 2) * 1))` ます。これはもちろん、 `6` とされます。これがであることをするには、 `3! == 3 * 2 * 1 == 6`。これむに、のとにどのようにするかをにしていることをしてください。

`if n == 0: return 1` の `if n == 0: return 1` ます。これは、をさないためです。ベースケースはにです。1つもなければ、あなたはにります。そうすれば、なくとも1つのケースがあるり、なだけくのケースをつことができます。たとえば、のうにに `factorial` をくことができます。

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

また、のケースがあるかもしれませんが、しく、にするのがしいがいのので、そのケースにはりません。

「」びしをつこともできます。えは、のうにされる **フィボナッチ** をえる。

- が `0` の、えは `0` です。
- が `1` の、えは `1` です。
- それの、えは `2` つのフィボナッチのです。

これはのようになります。

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

は `factorial(3)` でったようにこのをにしません、 `fib(5)` なりはの ににします

```
(
    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)
        )
    )
)
)
```

これはもちろん5される $(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))$ ます。

さて、いくつかののをりげてみましょう。

- **テールコール**は、にをすにされるのであるびしです。らかにするために、 `return foo(n - 1)` はテールコールですが、 `return foo(n - 1) + 1` はのであるため `return foo(n - 1) + 1` ません。
- テールコール TCOは、のをにらすです。
- テールコール TCEは、なしでできるへのテールコールのです。 TCEはTCOのです。

テールコールのは、さまざまなでちます。

- インタプリタは、によってされるメモリののをにえることができます。コンピュータにのメモリがないので、のなびしは **スタックのオーバーフローにつながります**。
- インタプリタは、 **スタックフレームスイッチ**のをらすことができます。

Pythonには、[さまざまな](#)でされた TCOはありません。したがって、このをちかすためにはのがである。のは、ユースケースによってなります。いくつかのでは、`factorial`と`fib`は、のようにコードににできます。

```
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
```

これは、をでするもなですが、よりなではむしろしくなります。

もうつのなツールはPythonの[lru_cache](#)デコレータで、なのをらすためにできます。

Pythonでのをけるについてえましたが、いつをうべきですかえは「しばしば」ではありません。すべてのはりしできます。どのようにしてうのかをするだけのです。ただし、がになることはです。Pythonでは、されるがびしのかかなりのをきこさないにがです。

がのあるなら、はSchemeやHaskellなどのをすることをおめします。そのようなでは、がずっとです。

フィボナッチシーケンスののは、Pythonでをし、でlruキャッシュをするをしています、ベースケースごとに2つのびしをうため、ながあります。へのびしのは n にします。

ではなく、よりなでは、をします。

```
def fib(n):
    if n <= 1:
        return (n, 0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

しかし、それはのペアをすがあります。これは、いくつかのがにからあまりられないことをしています。

をうツリー

のツリーがあるとします。

```
root
- A
  - AA
  - AB
```

```
- B
  - BA
  - BB
    - BBA
```

さて、もしのをすべてしたいのであれば、なforループでこれをうことができます。々は、ノードののをす `get_name()` と、ツリーのえられたノードのすべてのサブノードのリストをす `get_children()` と、 `get_root()` を `get_root()` て、ルートノードをします。

```
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
```

これはうまくいっていますが、サブノードがのサブノードをっていたらどうなりますかこれらのサブノードにはさらにくのサブノードがあるかもしれません。これをするは、のです。

```
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
```

しないで、すべてのノードのフラットなリストをすことをおめします。これは、ローリングリストをパラメータとしてすことでできます。

```
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']
```

の

なのさにはがあります。これはPythonのにします。にすると、`RuntimeError`がします。

```
RuntimeError: Maximum Recursion Depth Exceeded
```

このエラーのとなるプログラムのをにします。

```
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!
```

さをするには、のようにします。

```
sys.setrecursionlimit(limit)
```

のコマンドをすると、のをできます。

```
sys.getrecursionlimit()
```

たちのしいでとじをする

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

Python 3.5からののは、RuntimeErrorからしたRecursionErrorです。

テール . い

からされるのはびしだけの、とばれます。

をしてかれたカウントダウンのをにします。

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

をしてうことができるは、をしてうこともできます。をとてかれたfind_maxのバージョンです

```
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)
```

テイルは、Pythonコンパイラがのびしのをしないため、Pythonではいとみなされます。このようなソリューションは、のなソリューションよりもくのシステムリソースをします。

スタックイントロスペクションによるテール

デフォルトでは、Pythonのスタックは1000フレームをえることはできません。これは、よりい `sys.setrecursionlimit(15000)` をすることによってできますが、このメソッドはよりくのメモリをします。わりに、スタックイントロスペクションをしてテールをすることもできます。

```
#!/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
```

をするために、`@tail_call_optimized` デコレータをしてをびすことができます。のデコレータをしたなのをいくつかします。

```
@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.
```

フィボナッチ

```
@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.
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/1716/>

165:

Examples

py2app

py2appフレームワークをするには、まずそれをインストールする必要があります。これをうには、をき、のコマンドをします。

```
sudo easy_install -U py2app
```

また、のようにパッケージを`pip`することもできます

```
pip install py2app
```

に、Pythonスクリプトのセットアップファイルをします。

```
py2applet --make-setup MyApplication.py
```

セットアップファイルのをみにわけてします。これがデフォルトです

```
"""
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'],
)
```

アイコンファイルをするにはこのファイルのが`.icn`であるがあります、アプリケーションとしてイメージをめるには、のようにオプションをします。

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

にこれをにします

```
python setup.py py2app
```

スクリプトがされ、したアプリケーションがdistフォルダにあります。

さらにカスタマイズするには、のオプションをします。

optimize (-O)	optimization level: -O1 for "python -O", -O2 for "python -OO", and -OO to disable [default: -OO]
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

[ここ](#)からcx_Freezeをインストールしてください

フォルダをし、そのディレクトリからのコマンドをします。

```
python setup.py build
sudo python setup.py install
```

あなたのpythonスクリプトのためのしいディレクトリをし、のでじディレクトリに"setup.py"ファイルをしします

```
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)])
```

ターミナルからsetup.pyをしします

```
python setup.py bdist_mac
```

El Capitanでは、これをルートとしてし、SIPモードをにするがあります。

オンラインでをむ <https://riptutorial.com/ja/python/topic/2026/>

166:

は[PEP 435](#)によってバージョン3.4のPythonにされました。

Examples

のPython 2.4から3.3

はPython 3.4からPython 2.4からPython 3.3までバックポートされています。これはPyPIから[enum34](#)バックポートをできます。

```
pip install enum34
```

のはPython 3.4でのとじです

```
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
```

はりしです

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3

[c for c in Color] # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/947/>

167: な

き

Pythonは、との、べき、など、なをでします。モジュールすべてのPythonバージョンにまれていますは、ルート、などのをします。

とそのメタクラス

`numbers`モジュールには、のメタクラスがまれています。

サブクラス	。	。インテグラ		リアル。	<code>numbers.Complex</code>
ブール	✓	✓		✓	✓
<code>int</code>	✓	✓		✓	✓
フラクシオン。フラクシオン	✓	-		✓	✓
<	✓	-		-	✓
	✓	-		-	✓
<code>decimal.Decimal</code>	✓	-		-	-

Examples

```
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
```

なみわせみみ

- `int`と`int`え`int`

- int と float $\rightarrow$ float
- int と complex $\rightarrow$ complex
- float と float $\rightarrow$ float
- float と complex $\rightarrow$ complex
- complex かつ complex $\rightarrow$ complex

+ は、リスト、およびタプルをするためにもされます。

```
"first string " + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

```
a, b = 1, 2

# Using the "-" operator:
b - a                                # = 1

import operator                      # contains 2 argument arithmetic functions
operator.sub(b, a)                   # = 1
```

なみわせみみ

- int と int $\rightarrow$ int
- int と float $\rightarrow$ float
- int と complex $\rightarrow$ complex
- float と float $\rightarrow$ float
- float と complex $\rightarrow$ complex
- complex かつ complex $\rightarrow$ complex

```
a, b = 2, 3

a * b                                # = 6

import operator
operator.mul(a, b)                   # = 6
```

なみわせみみ

- int と int $\rightarrow$ int
- int と float $\rightarrow$ float
- int と complex $\rightarrow$ complex
- float と float $\rightarrow$ float
- float と complex $\rightarrow$ complex
- complex かつ complex $\rightarrow$ complex

*は、リスト、およびタプルのリシにもされます。

```
3 * 'ab' # = 'ababab'
3 * ('a', 'b') # = ('a', 'b', 'a', 'b', 'a', 'b')
```

のオペランドがの、Pythonはをいます。 Pythonののは、Python 2.xと3.xからされています [Integer Division](#) もしてください。

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x 2.7

Python 2では、'/'のはとのにします。

```
a / b # = 1
a / c # = 1.5
d / b # = -2
b / a # = 0
d / e # = -1
```

aとbがintであるため、はintます。

はにめられます。

cはfloatなのでa / cはfloatです。

モジュールをすることもできます。

```
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

フロートがな

```
from __future__ import division # applies Python 3 style division to the entire module
a / b # = 1.5
a // b # = 1
```

さてあなたはモジュールにしたいくない

```
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
```

しませんがななど、TypeErrorがすることがあります。

```
float(a) / b      # = 1.5
a / float(b)      # = 1.5
```

Python 2.x 2.2

Python 2の'/'は、`int`ではなく、`float`をします。

```
a // b      # = 1
a // c      # = 1.0
```

Python 3.x 3.0

Python 3では、`/`は`'/'`ではなく`'/'`をします。`//`はフロアをい、をします。

```
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
```

なみわせみみ

- int と int Python 2では int をえ、Python 3では float をす
- int と float え float
- int と complex え complex
- float と float え float
- float と complex え complex
- complex かつ complex え complex

は、 [PEP 238](#)をしてください。

```
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
```

みみ`pow`と`math.pow`のいは、みみ`pow`が3つのをけることができることです。

```
a, b, c = 2, 3, 2

pow(2, 3, 2)          # 0, calculates (2 ** 3) % 2, but as per Python docs,
                      # does so more efficiently
```

な

`math.sqrt(x)`は、`x`をします。

```
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)
```

キューブルートなどののルートをするには、ルートののにをげます。これは、またはのいずれかをってうことができます。

```
import math
x = 8
math.pow(x, 1/3) # evaluates to 2.0
x**(1/3) # evaluates to 2.0
```

`math.exp(x)` `e ** x`します。

```
math.exp(0) # 1.0
math.exp(1) # 2.718281828459045 (e)
```

`math.expm1(x)` `e ** x - 1`します。`x`がさい、これは`math.exp(x) - 1`よりもはるかにれたをえます。

```
math.expm1(0)          # 0.0

math.exp(1e-6) - 1     # 1.0000004999621837e-06
math.expm1(1e-6)       # 1.0000005000001665e-06
# exact result         # 1.000000500000166666708333341666...
```

デフォルトでは、`math.log`はのをします。オプションで、ベースを2としてすることもできます。

```
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)
```

`math.log`のなバリエーションは、さまざまにします。

```
# 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)
```

めみ

アプリケーションでは、のようなコードをつがります。

```
a = a + 1
```

または

```
a = a * 2
```

これらののためのなショートカットがあります。

```
a += 1
# and
a *= 2
```

インプレースをうには、`'='`のののをできます。

- `--`をする
- `+=`をのにインクリメントする
- `*=`をにじる
- `/=`をなにする
- `//=` `floor`はをのにするPython 3
- `%=`のモジュラスをのにす
- `**=`でパワーアップする

ビット`^`、`|`などには、のがします。

```
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
```

`math.hypot(x, y)` は、 $(0, 0)$ から (x, y) までのベクトルまたはユークリッドの長さでもあらうにしてください。

2つの (x_1, y_1) と (x_2, y_2) のユークリッドをするには、のように `math.hypot` をできます

```
math.hypot(x2-x1, y2-y1)
```

ラジアン $\rightarrow$ と $\rightarrow$ ラジアンからするには、それぞれ `math.degrees` と `math.radians` します

```
math.degrees(a)
# Out: 57.29577951308232

math.radians(57.29577951308232)
# Out: 1.0
```

モジュラス

のくのとに、Pythonはモジュラスのに`%`をします。

```
3 % 4      # 3
10 % 2     # 0
6 % 4      # 2
```

または、`operator` モジュールをして

```
import operator

operator.mod(3, 4)      # 3
operator.mod(10, 2)     # 0
operator.mod(6, 4)      # 2
```

のをすることもできます。

```
-9 % 7      # 5
9 % -7      # -5
-9 % -7     # -2
```

とモジュラスのをつけるがあるは、`divmod` をショートカットとしてできます。

```
quotient, remainder = divmod(9, 4)
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

オンラインでなをむ <https://riptutorial.com/ja/python/topic/298/> な

168:

Examples

の

Python 3では、`print`はキーワードではなくです。

```
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
```

また、いくつかのパラメータをして`print`することもでき`print`。

```
print(foo, bar, baz)
# out: 1 bar 3.14
```

のパラメータを`print`するのは、`+`

```
print(str(foo) + " " + bar + " " + str(baz))
# out: 1 bar 3.14
```

しかし、のパラメータをするために`+`をするはがですが、パラメータのタイプがじであるがあります。`"bar"1`をえて`3.14`にしようとするため、のを`string`にキャストせずにしようするとエラーになります。

```
# Wrong:
# type:int str float
print(foo + bar + baz)
# will result in an error
```

これは、`print`がにされるためです。

```
print(4 + 5)
# out: 9
print("4" + "5")
# out: 45
print([4] + [5])
# out: [4, 5]
```

その、+をすると、ユーザーがのをみすのににちます。のでは、をみやすくなっています。

のスク립トはこれをしていいます

```
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))
```

のendパラメータをすると、printがをにしないようにすることができます。

```
print("this has no newline at the end of it... ", end="")
print("see?")
# out: this has no newline at the end of it... see?
```

ファイルにきむは、パラメータfileとしてすことができますfile

```
with open('my_file.txt', 'w+') as my_file:
    print("this goes to the file!", file=my_file)
```

これはファイルにく

パラメータ

なるテキストののことができます。printにはいくつかのパラメータがあります。

sep のにをします。

カンマやそののでられたのリストをするがありますか

```
>>> print('apples','bannas','cherries', sep=', ')
apple, bannas, cherries
>>> print('apple','banna','cherries', sep=', ')
apple, banna, cherries
>>>
```

のend endのものをう

endがなければ、すべてのprint()はをき、ののにします。2つのをってもしないようにの"のをす、またはのを2にすることができます。

```
>>> print("<a", end=''); print(" class='jldn'" if 1 else "", end=''); print("/>")
<a class='jldn'/>
>>> print("paragraph1", end="\n\n"); print("paragraph2")
paragraph1

paragraph2
>>>
```

`file` を `sys.stdout` のにります。

これで、`stdout`、`file`、または `StringIO` のいずれかにテキストをできます。それがファイルのようにれた、それはファイルのようにします。

```
>>> 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

>>>
```

にストリームを `flush` する4のパラメータ `flush` あります。

オンラインでをむ <https://riptutorial.com/ja/python/topic/1360/>

169: スコープとバインディング

- グローバル `a`、`b`、`c`
- な `a`、`b`
- `x = かはx` をする
- `x`、`y = something` は `x` と `y` をする
- `x += something` は `x` をバインドします。にのすべての "`op =`"
- `del x` は `x` をする
- `x` は `か` ので binds `x`
- `か` を `x` とする binds `x`
- `Ex` と `Ex` を `く` ブロックにをバインドする

Examples

グローバル

Pythonでは、`の`は、`の`にされるや、`の`バインディングオカレンスがされるにり、ローカルとなされます。さもなくば、そのようなバインディングは、グローバルスコープまでのみでされます。これは、`が`してされなくてもてはまります。

```
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
```

、スコープの`の`は、`じ`のをすべてシャドウします。

```
x = 'Hi'
```

```
def change_local_x():
    x = 'Bye'
    print(x)
change_local_x() # prints Bye
print(x) # prints Hi
```

を`global`すると、スコープのりののでは、モジュールのトップレベルでへのりてがわれます。

```
x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x() # prints Bye
print(x) # prints Bye
```

`global`キーワードは、プログラムのトップレベルではなくモジュールのトップレベルでがわれることをします。のモジュールでは、モジュールのへののドットきアクセスがです。

すると、`x`がにしてローカルであるかどうかをるためには、をむがあります。

1. `global x`がかった、`x`はグローバルです
2. `nonlocal x`がかった、`x`はみにし、ローカルでもグローバルでもありません
3. `x = 5`または`for x in range(3)`またはそののバインディングをつけた、`x`はローカルです
4. 其の`x`は、いくつかのみスコープスコープ、グローバルスコープ、またはみにします。

ローカル

でがバインドされている、デフォルトではでのみアクセスできます。

```
def foo():
    a = 5
    print(a) # ok

print(a) # NameError: name 'a' is not defined
```

フローはスコープにをえませんを`except`が、まだりてられていないへのアクセスはエラーです

```
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
```

のがりてある`for`などのループ、およびりて`a += 5`

Python 3.x 3.0

Python 3は**nonlocal**という新しいキーワードをしました。nonlocalキーワードはスコープオーバーライドをスコープにします。 [PEP 3104](#)でそれにするすべてをむことができます。これは、いくつかのコードでもよくされています。ものの1つは、インクリメントできるをすることです。

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

このコードをしようとすると、**num**がものにりてられるにされるため、**UnboundLocalError**がされます。ミックスにローカルをしましょう

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

にnonlocalは、スコープのにりてることができますが、グローバルスコープはりてられません。したがって、counterでnonlocalをすることはできません。なぜなら、グローバルスコープにしようとするからです。してみると、すぐにSyntaxErrorられSyntaxError。わりに、ネストしたでnonlocalをするがあります。

ここにすは、ジェネレータをしてするがよいことにしてください。

バインディングの

```
x = 5
x += 7
for x in iterable: pass
```

のステートメントはそれぞれバインディングオカレンスです `x` 5されるオブジェクトにバインドされます。このステートメントがにある、`x`はデフォルトでローカルになります。バインディングステートメントのリストについては、「」をしてください。

をするときにクラススコープをスキップする

クラスはにローカルスコープをちますが、クラスのはをルックアップするときはそのスコープをしません。ラムダはなので、スコープをとってがされているので、これはいくつかのくべきにつな

がります。

```
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
```

このスコープのみになじみのないユーザーは、`b`、`c`、および`e`が`class`をすることをされるかもしれません。

PEP 227から

クラススコープのにはアクセスできません。はのスコープでされます。ネストされたスコープのチェーンでクラスがすると、プロセスはクラスをスキップします。

とバインディングにするPythonのから

クラスブロックでされるのは、クラスブロックにされます。メソッドのコードブロックにはされません。これは、スコープをしてされるため、これにはとジェネレータのがまれます。これは、がすることをします。

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

このでは、Martijn Pietersによるこののをしています。このには、こののいがまれています。

del コマンド

このコマンドには、いくつかのしたのがあります。

```
del v
```

`v`がの、コマンド`del v`はをスコープからします。えは

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'x' is not defined
```

`del`はなのであることにしてください。のでにしないり `nonlocal` または `global` をして、`del v`は`v`をのスコープにローカルにします。スコープで`v`をするは、`del v`ステートメントとじスコープで`nonlocal v`または`global v`します。

のすべてにおいて、コマンドのはデフォルトのですが、によってされません。クラスはこのをにするでされるかもしれません。

`del v.name`

このコマンドは、`v.__delattr__(name)`びしをトリガーします。

は、`name`にすることです。えは

```
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]`

このコマンドは、`v.__delitem__(item)`びしをトリガします。

は、オブジェクト`v`によってされたマッピングに`item`がしていないことです。えは

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) # out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
```

`del v[a:b]`

これはに`v.__delslice__(a, b)`びします。

は、のものですが、ののわりにのであるスライスをします。えは

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) # out: [0, 3, 4]
```


ガベージコレクション`del`コマンドもしてください。

ローカルスコープとグローバルスコープ

ローカルおよびグローバルスコープとはですか

コードのあるでアクセスなすべてのPythonは、ローカルスコープまたはグローバルスコープのいずれかにあります。

には、ローカルスコープにはのでされたすべてのがまれ、グローバルスコープにはのでされたがまれます。

```
foo = 1 # global

def func():
    bar = 2 # local
    print(foo) # prints variable foo from global scope
    print(bar) # prints variable bar from local scope
```

どのがどのスコープにあるかをverることができます。みみ`locals()`および`globals()`はスコープをとしてします。

```
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
```

のはどうなりますか

```
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
```

グローバルをするには、keyword `global` します。

```
foo = 1

def func():
    global foo
    foo = 2 # this modifies the global foo, rather than creating a local variable
```

スコープは、にしてされています

それがすることは、のはをしてグローバルにすることはできず、そのはをしてグローバルにすることはできません。

```
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)
```

に、オボジト

```
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
```



にはくのレベルのがネストされているかもしれませんが、いずれかののにはそのとグローバルスコープのローカルスコープが1つしかありません。スコープはありません。

```
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 と nonlocal Python 3のみ

のキーワードは、のにローカルではないへのきみアクセスをするためにされます。

global キーワードは、がグローバルとしてわれるべきであることをします。

```
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
```

、Python 3でな nonlocal [Nonlocal Variables](#)は、ローカルスコープからののにローカルスコープにローカルをします。

nonlocal の[Python ドキュメント](#)から

ローカルステートメントは、リストされたが、グローバルをくもいみスコープののにバインドされたをするようにします。

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!
```

オンラインでスコープとバインディングをむ <https://riptutorial.com/ja/python/topic/263/スコープとバインディング>

170:

- `map`、`iterable` [、 * `additional_iterables`]
- `future_builtins.map`、`iterable` [、 * `additional_iterables`]
- `itertools.imapfunction`、`iterable` [、 * `additional_iterables`]

パラメーター

パラメータ	
	マッピングのためのiterablesがあるのとじくらくのパラメータを あります
りしな	は のみ のにされ、
* additional_iterables	iterableをしてください。ただし、きなだけく オプション、

`map` をってうことができるすべてのことは、 `comprehensions` をもってうこともできます。

```
list(map(abs, [-1,-2,-3]))    # [1, 2, 3]
[abs(i) for i in [-1,-2,-3]]  # [1, 2, 3]
```

のiterableがあるは、 `zip` が必要です。

```
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]
```

リストはであり、くの`map`よりもくなるがあるため、がなはのアプローチのをテストします。

Examples

`map`、`itertools.imap`、`future_builtins.map`のない

マップは、プログラミングにされるPythonビルトインのでもなものです。 `map()` は、されたをな
のにします。

```
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>
```

future_builtins モジュールには、Python 3のmapがまかれて future_builtins ます。

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>
```

あるいは、Python 2では itertools imap をってジェネレータをることができます

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>
```

をに list にして、Python 2とPython 3のいをりくことができます

```
list(map(len, names))
# Out: [4, 5, 6]
```

map() は、の リスト または ジェネレータ にきえることができます

```
[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>
```

iterable ののマッピング

たとえば、のをることができます。

```
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]
```

はリストのマッピングもサポートしています

```
map(lambda x:x*2, [1, 2, 3, 4, 5])
# Out: [2, 4, 6, 8, 10]
```

のをパーセンテージにする

```
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]
```

ドルをユーロにするレートを

```
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`は、のパラメータを`lambda`をうのではなく`map`でうか、カスタマイズされたをるかのようにするなです。

なるiterablesののマッピング

えは、のイテラブルの`i`ののをする

```
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]
```

Pythonのバージョンにじて、のiterableが`map`されたのはなります。

- このは、iterableとじのパラメータをるがあります。

```
def median_of_three(a, b, c):
    return sorted((a, b, c))[1]

list(map(median_of_three, measurement1, measurement2))
```

`TypeError: median_of_three missing 1な 'c'`

```
list(map(median_of_three, measurement1, measurement2, measurement3, measurement3))
```

`TypeError: median_of_threeは3つのをるが、4つがえられた`

Python 2.x 2.0.1

- `map` 1つのiterableがまだにはされていないが、にされたiterableから`None`をするり、マッピン

グはりします。

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
```

TypeError - 'int'および'NoneType'のサポートされていないオペランドタイプ

- `itertools.imap`および`future_builtins.map`つの`future_builtins.map`がするとすぐにマッピングがします。

```
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

- 1つのがするとすぐにマッピングはします。

```
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]
```

マップをったのとして **"None"**をう**Python 2.x**のみ

```
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 'NoneType'オブジェクトはびしではありません`

しかし、のをるためのがあります

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# Out: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

シリーズおよびパラレルマッピング

`map`はビルトインです。つまり、`'import'`ステートメントをすることなくどこでもできます。5を
ると、`pretty print``import pprint`をするに`import`をしなければならないことがわかります。したがっ
て、`pprint`はみみではありません

シリーズマッピング

この、イテラブルのは、マッピングのとしてにされます。これは、マッピングするが1つしかなく
、マッピングがのをとするにします。

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
```

は

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

2

```
print(list(map(len, insects))) # the len function is executed each item in the insect list
```

は

```
[3, 3, 6, 10]
```

パラレルマッピング

この、マッピングのは、すべてのiterableiterableから1つからにきされます。したがって、されるイテラブルのは、がとするのとしなければなりません。

```
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)
```

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)))
```

は

```
TypeError: len() takes exactly one argument (4 given)
```

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)))
```

は

```
TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

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)))
```

は

```
['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']
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/333/>

171:

Examples

`input`と`raw_input`をすると、

Python 2.x 2.3

`raw_input`は、ユーザーがテキストをするのをってから、そのをとします。

```
foo = raw_input("Put a message here that asks the user for input")
```

のでは、`foo`はユーザーがしたものをします。

Python 3.x 3.0

`input`はユーザーがテキストをするのをってから、をとします。

```
foo = input("Put a message here that asks the user for input")
```

のでは、`foo`はユーザーがしたものをします。

の

Python 3.x 3.0

Python 3では、はのをしています

```
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

Python 2では、`print`はもともとはのようなステートメントでした。

```
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で`from __future__ import print_function`をすると、ユーザーはPython 3コードと同じ`print()`をできます。これは、Python 2.6でのみです。

ユーザにをす

```
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)
```

それをうには

```
user_number = input_number("input a number: ", "that's not a number!")
```

または、「エラーメッセージ」をしたくないは、のようにします。

```
user_number = input_number("input a number: ")
```

になしのをす

Python 2.x 2.3

Python 2.xでは、`print`でをけるには、`print`をカンマでわさせます。にスペースがされます。

```
print "Hello,",
print "World!"
# Hello, World!
```

Python 3.x 3.0

Python 3.xでは、`print`には、されたのにされるオプションの`end`パラメータがあります。デフォルトではなので、これにします

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

しかし、あなたはのをすことができます

```
print("Hello, ", end="")
print("World!")
```

```
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

をよりにするには、 `sys.stdout.write` をします。

```
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

stdin からみむ

Python プログラムは [UNIX パイプライン](#) からみむことができます。ここでは、 `stdin` からみむのなをします

```
import sys

for line in sys.stdin:
    print(line)
```

`sys.stdin` はストリームであることにしてください。これは、ストリームがしたときにのみ `for` ループがすることをします。

のように、のプログラムのを Python プログラムにパイプすることができます

```
$ cat myfile | python myprogram.py
```

このでは、 `cat myfile` は `stdout` する `unix` コマンドです。

あるいは、 [fileinput モジュール](#) をうとです

```
import fileinput
for line in fileinput.input():
    process(line)
```

ファイルからの

はファイルからみることもできます。ファイルは、 `open` をしてることができます。 `with <command> as <name> 'コンテキストマネージャ' とはれますと with <command> as <name> すると、 open をしてファイルのハンドルをにれやすくなります。`

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

これにより、コードのがブロックをれると、ファイルはにじられます。

ファイルはなるモードでくことができます。のでは、ファイルはみりとしてかれています。のファイルのみりでくには`r`します。あなたがバイトとしてそのファイルをもには、`rb`い`rb`。のファイルにデータをするには、`a`ます。`w`をしてファイルをするか、じののファイルをきします。`r+`をすると、みきのでファイルをくことができます。`open()`1はファイルであり、2はモードです。モードをのままにすると、デフォルトは`r`になります。

```
# 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']
```

ファイルのサイズがさい、ファイルをメモリにみむことはです。ファイルがにきいは、またはチャックでみみ、をじループでするほうがよいがあります。それをうには

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

ファイルをむときは、オペレーティングシステムのをしてください。`for line in fileobj`にそれら`for line in fileobj`ますが、のように、みまれたの`strip()`をびすことはにです。

のでは、いているファイル `fileobj` はにファイルののをしています。ファイルハンドルがにかけると、ファイルハンドルはファイルのの、つまり0します。ファイルハンドルは`tell`つ`tell`のをすることができます

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

すべてのコンテンツをみむと、ファイルハンドラのはファイルのにポイントされます

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

ファイルハンドラのは、なものにすることができます

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

また、したにファイルからのさを見ることもできます。これをうには、`read()`をします。なしで`read()`をびすと、ファイルのわりまでみみます。をすと、モードにじてそのバイトまたはがみまれますそれぞれ`rb`および`r`。

```
# 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()
```

とバイトのいをすには

```
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'>
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/266/>

172: なデータint、float、str、tuple、frozensets

Examples

の々のをりてることはできません

```
foo = "bar"
foo[0] = "c" # Error
```

なは、するとできません。

タプルの々のメンバーはりてできません

```
foo = ("bar", 1, "Hello!",)
foo[1] = 2 # ERROR!!
```

したタプルメンバはりてなので、2はエラーをします。タプルののために。

Frozensetはであり、りてられません

```
foo = frozenset(["bar", 1, "Hello!"])
foo[2] = 7 # ERROR
foo.add(3) # ERROR
```

2のは、されたfrozensetメンバがりてられないのでエラーをします。frozensetsはメンバーをできるをサポートしていないため、3はエラーをします。

オンラインでなデータint、float、str、tuple、frozensetsをむ

<https://riptutorial.com/ja/python/topic/4806/なデータ-int-float-str-tuple-frozensets->

Examples

は、オブジェクトのになくオブジェクトにしてアクションをします。これは、に、クラスをし、すべてがじシングネチャをつメソッドをする2つのサブクラスをつことによってされます。これらのオブジェクトをするのまたはメソッドは、のとなるオブジェクトのタイプになく、チェックをにうなく、じメソッドをびすことができます。クラスXがクラスYをするオブジェクトでは、Yはスーパークラスまたはベースクラスとばれ、Xはサブクラスまたはクラスとばれます。

```
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)
```

私たちはこのをるべきです

し
4
6.0

がなければかこるでしょうか

ポリモーフィズムがなければ、オブジェクトにしてアクションをしてしいびしをするにチェック
がながあります。のカウンタのは、のコードとじタスクをしますが、をしない、 `get_area()` はよ
りくのをうががあります。

```
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)

```

私たちはこのをるべきです

4
6.0

な

カウンターのでされるクラスは "しいスタイル"クラスであり、Python 3がされているはにオブジェクトクラスからされています。はPython 2.xと3.xのですが、input_objのためPython 2.xインタプリタでされるとコードはをさせます。らはにりてられていないされないので、その、オブジェクトからされていないではなく、クラスの「インスタンス」をします。

ダックタイピング

Pythonでなダックタイピングのでされていない。これは、クラスにじメソッドがまれているり、Pythonインタプリタはそれらをしないことをします。にびしのチェックだけがわれるからです。

```

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)

```

はのとおりで。

クアアアック

アヒルにはいのがあります。
そのはアヒルをする。
そのはからをとり、それをします。

オンラインでをむ <https://riptutorial.com/ja/python/topic/5100/>

174:

Examples

リストのリスト

2をするいは、リストのリストです。このようなもの

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

ここでは、のリスト`lst`、それに3つのことをしています。のものは`[1,2,3]`、2のものは`[4,5,6]`、3のものは`[7,8,9]`です。これらのリストにアクセスするには、リストののにアクセスするのとじがあります。

```
print (lst[0])
#output: [1, 2, 3]

print (lst[1])
#output: [4, 5, 6]

print (lst[2])
#output: [7, 8, 9]
```

これらのリストのそれぞれのにじでアクセスできます。

```
print (lst[0][0])
#output: 1

print (lst[0][1])
#output: 2
```

ここでは、`[]`ののは、そのにあるリストをすることをします。のでは、`0`をして、`[1,2,3]` `0`のリストをすることをしています。2のの`[]`は、そののをのリストからすることをします。このでは、`0`と`1`のをして、リストの`0`のは`1`で、`1`のは`2`

これらのリストのもにできます。

```
lst[0]=[10,11,12]
```

リストは`[[10,11,12],[4,5,6],[7,8,9]]`です。このでは、のリストをくしいリストにしました。

```
lst[1][2]=15
```

リストは`[[10,11,12],[4,5,15],[7,8,9]]`です。このでは、のリストの1つのでのをしました。に、`1`のリストになって、そののを`2`にしました。2になりました。は`6`で、は`15`です。

リストのリストのリスト

このはできます。ここに3があります

```
[[ [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]]]
```

おそろくらかなように、これはみにくいです。バックスラッシュをしてさまざまなディメンションをします。

```
[[ [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]]]
```

このようなリストをネストすることで、のにすることができます。

アクセスは2Dにしています

```
print(myarray)
print(myarray[1])
print(myarray[2][1])
print(myarray[1][0][2])
etc.
```

もです。

```
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.
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/8186/>

175: アクセス

- `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

ドットをしたアクセス

サンプルクラスをけましょう。

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

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

Pythonでは、ドットをしてクラスのタイトルにアクセスできます。

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

がしない、Pythonはエラーをスローします

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

セッター、ゲッタープロパティ

データのカプセルのために、のからがられるをつことがなもあります。また、に、そのでどのがされるのかがあります。このをするなは、getterまたはsetterとばれるメソッドをすることです。

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

のでは、タイトルとをむしいブックをするとどうなるかをにできます。ライブラリにするすべてののにとタイトルがまれているは、ゲッターとセッターをスキップしてドットをできます。しかし、がないがあり、に「」をしたいとします。あるいは、にのがいて、のリストをすがある。

この、`author`のgetterとsetterをできます。


```

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

```

これはおめしません。

1つのは、キャッチがあるということです。パブリックをち、メソッドをたないクラスをしたとしましょう。々はすでにこれをくしており、のようなコードをいています。

```

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

```

たちにはがあります。はではないので Pythonは、こののプロパティーをするソリューションをしています。プロパティをするメソッドは、ヘッダのに@propertyでされています。たちがセッターとしてさせたいメソッドは、そのに@ attributeName.setterでされています。

これをにいて、しいクラスをしました。

```

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

```

、Pythonでは、じとなるのパラメータをつのメソッドをつことはできません。しかし、このPythonではデコレータがされているため、これがです。

コードをテストすると

```

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

```

オンラインでアクセスをむ <https://riptutorial.com/ja/python/topic/4392>/アクセス

176:

Examples

キーの

キーがするかどうかかわからないのは、`dict.get` メソッドをすることをおめします。 `key` がつかからないのは、デフォルトをすることが出来ます。 のメソッド `dict[key]` は `KeyError` をさせます。

やるよりも

```
def add_student():
    try:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

う

```
def add_student():
    students['count'] = students.get('count', 0) + 1
```

のりえ

2つののをりえるには、タプルをできます。

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

テストをう

Pythonは、のオブジェクトをテストのためにブールににするので、なりしてください。

```
# 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

```

これはに、よりみやすいコードをし、しないをうときにはるかにです。

Falseにされるのは[こちらをクリックしてください](#)。

しないコードのをけるための "**__main__**" のテスト

コードをするに、びしプログラムの `__name__` をテストすることをお `__name__` ます。

```

import sys

def main():
    # Your code starts here

    # Don't forget to provide a return code
    return 0

if __name__ == "__main__":
    sys.exit(main())

```

このパターンをすると、コードがどおりにされることがされます。たとえば、ファイルをしにするとします。

```
python my_program.py
```

ただし、のプログラムでファイルを `import` するたとえば、ライブラリとしてファイルをきむなど、がられます。ファイルを `import` すると、 `__main__` トラップによってコードがせずされることがなくなります。

```

# 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()

```

オンラインでをむ <https://riptutorial.com/ja/python/topic/3070/>

177: クラスabc

Examples

ABCMeta メタクラスの

クラスはされるクラスですが、のメソッドのはけ、サブクラスがするがあるメソッドシグネチャのみをします。

クラスは、のインタフェースのとに、メソッドのをとせずに、クラスをレベルでしてするのにです。

クラスをするアプローチの1つは、クラスメソッドをスタブアウトし、アクセスされたは `NotImplementedError` をさせることです。これにより、クラスがメソッドにオーバーライドすることなくアクセスすることがされます。そのようです

```
class Fruit:

    def check_ripeness(self):
        raise NotImplementedError("check_ripeness method not implemented!")

class Apple(Fruit):
    pass

a = Apple()
a.check_ripeness() # raises NotImplementedError
```

このようにクラスをすると、オーバーライドされていないメソッドがにされることがなくなり、クラスでメソッドをすることをしますが、をすることはありません。 `abc` モジュールをすると、クラスとのクラスメソッドをオーバーライドできないときに、クラスがインスタンスされるのをぐことができます。

```
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.
```

にサブクラスしてオーバーライドすることができます

```
class Subclass(ABCClass):
    def virtual_method_subclasses_must_define(self):
        return
```

なぜ/ **ABCMeta** と **@abstractmethod** のい

クラスABCは、クラスがクラスからのメソッドをすることをします。

これがどのようにするのか、なぜそれをすべきかをするために、Van Rossumがしめるをてみましょう。すべてのクラスであるがある2つのメソッドジョークパンチラインをつBaseクラス "MontyPython"があるとしましょう。

```
class MontyPython:
    def joke(self):
        raise NotImplementedError()

    def punchline(self):
        raise NotImplementedError()

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

オブジェクトをインスタンスして2つのメソッドをびすと、 `punchline()` メソッドで `punchline()` どりエラーがします。

```
>>> sketch = ArgumentClinic()
>>> sketch.punchline()
NotImplementedError
```

しかし、これにより、エラーをさずにArgumentClinicクラスのオブジェクトをインスタンスすることができます。には、パンチラインをすまではエラーはしません。

これは、クラスABCモジュールをすることによってされます。じでこれがどのようにするかをてみましょう

```
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"
```

こので、なクラスからオブジェクトをインスタンスしようとする、すぐにTypeErrorがされます

o

```
>>> c = ArgumentClinic()
TypeError:
"Can't instantiate abstract class ArgumentClinic with abstract methods punchline"
```

この、クラスをさせてTypeErrorsをするのはです

```
class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"

    def punchline(self):
        return "Send in the constable!"
```

はオブジェクトをインスタンスするときにします

オンラインでクラスabcをむ <https://riptutorial.com/ja/python/topic/5442/クラス-abc->

Examples

Python スクリプトでをする

これはPythonスクリプトをし、されたについて、がまった、がどこでわるところ、がわるところ、そしてがわるところをします。

```
#!/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)
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/5370/>

179: の

Examples

CエクステンションをつHello World

のCソースファイルデモンストレーションので `hello.c` とぶは、 `hello` というのモジュールをします。このモジュールには、の `greet()` がまれています。

```
#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
```

`gcc` コンパイラでファイルをコンパイルするには、おみのターミナルでのコマンドをします。

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

にいた `greet()` をするには、じディレクトリにファイルをし、 `hello.py`

```
import hello          # imports the compiled library
```

```
hello.greet("Hello!") # runs the greet() function with "Hello!" as an argument
```

いているファイルを**C**のにす

いているファイルオブジェクトをPythonからCコードにします。

`PyObject_AsFileDescriptor`をしてファイルをファイルにすることができます

```
PyObject *fobj;
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0){
    return NULL;
}
```

ファイルをPythonオブジェクトにするには、 `PyFile_FromFd`し`PyFile_FromFd`。

```
int fd; /* Existing file descriptor */
PyObject *fobj = PyFile_FromFd(fd, "filename", "r", -1, NULL, NULL, NULL, 1);
```

C ++とBoostをしたC

これは、C ++とBoostをしたCのなです。

C ++コード

hello.cppにれられたC ++コード

```
#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)
        ;
}

```

これをPythonモジュールにコンパイルするには、Pythonヘッダとboostライブラリが必要です。このは、Python 3.4とgccをしてUbuntu 12.04でされました。Boostは多くのプラットフォームでサポートされています。Ubuntuの、なパッケージはをしてインストールされました

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

ソースファイルを.soファイルにコンパイルします。このファイルは、でPythonパスにあれば、モジュールとしてインポートできます。

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -
lboost_system -l:libpython3.4m.so
```

example.pyファイルのpythonコード

```

import hello

print(hello.get_hello())

h = hello.Hello("World hello!")
print(h.as_list(3))

```

に、 `python3 example.py` はのようになります

```

Hello world!
['World hello!', 'World hello!', 'World hello!']

```

オンラインでのをむ <https://riptutorial.com/ja/python/topic/557/>の

180: モジュール

Examples

めラウンド、フロア、セイル、トランケーション

ビルトイン `round` にえて、 `math` モジュールは `floor`、 `ceil`、 および `trunc` をします。

```
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`、 `round` に `float` します。

```
round(1.3) # 1.0
```

`round` にゼロかられてタイをります。

```
round(0.5) # 1.0
round(1.5) # 2.0
```

Python 3.x 3.0

`floor`、 `ceil`、 および `trunc` に `Integral` をしますが、 `round` は1つのでびされると `Integral` をします。

```
round(1.3)      # 1
round(1.33, 1)  # 1.3
```

`round` ブレークはもいにかってびついています。これは、のをするとき、よりきなにってします。

```
round(0.5) # 0
round(1.5) # 2
```

のとに、をにすることはできません。これにより、のめがするがあります。

```
round(2.675, 2) # 2.67, not 2.68!
```

のの**floor**、**trunc**、およびにする

PythonとC++とJavaはののためにゼロからめます。してください

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

`math.log(x)`、ベースえる e の x 。

```
math.log(math.e) # 1.0
math.log(1)      # 0.0
math.log(100)    # 4.605170185988092
```

のにより、`math.log`が1にいでわれるがあります。1にilogをにするには、1のとをする`math.log1p`します。

```
math.log(1 + 1e-20) # 0.0
math.log1p(1e-20)  # 1e-20
```

`math.log10`はログベース10にできます

```
math.log10(10) # 1.0
```

Python 2.x 2.3.0

2つのをすると、`math.log(x, base)`はえられた`base`つまり $\log(x) / \log(base)$ の x のを`math.log(x, base)` $\log(x) / \log(base)$ 。

```
math.log(100, 10) # 2.0
math.log(27, 3)   # 3.0
math.log(1, 10)   # 0.0
```

のコピー

Python 2.6では、`math.copysign(x, y)`は x のを y としてします。りはに`float`です。

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
```

のさの

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)
# Out: 4.47213595499958
```

をラジアンから/ラジアンにする

すべての`math`はラジアンをとするため、をラジアンにするがあります。

```
math.radians(45)          # Convert 45 degrees to radians
# Out: 0.7853981633974483
```

のすべてののは、をラジアンでします。したがって、それをにするがあります。

```
math.degrees(math.asin(1)) # Convert the result of asin to degrees
# Out: 90.0
```

、および

```
# 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"
```

`math.atan`とはに、2つのを`math.atan2`もあります。これは、しいをし、ゼロのとしをします。

```
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"
```

サイン、コサイン、タンジェント

```
# 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
```

`math`モジュールには、にされる2つのがまれています。

- `math.pi` - π
- `math.e` - e の

```
>>> from math import pi, e  
>>> pi  
3.141592653589793  
>>> e  
2.718281828459045  
>>>
```

Python 3.5では、とNaN「ではない」のがあります。を`float()`にすといういでもします。

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
```

Pythonののは、ターゲットのろにある "j"または "J"でされます。

```
1j          # Equivalent to the square root of -1.
1j * 1j     # = (-1+0j)
```

InfinityとNaN「ではない」

Pythonのすべてのバージョンでは、のようとNaN "not a number"をすることができます

```
pos_inf = float('inf')      # positive infinity
neg_inf = float('-inf')     # negative infinity
not_a_num = float('nan')    # NaN ("not a number")
```

Python 3.5では、された`math.inf`と`math.nan`もできます。

Python 3.x 3.5

```
pos_inf = math.inf
neg_inf = -math.inf
not_a_num = math.nan
```

は`inf`と`-inf`および`nan`としてされます。

```
pos_inf, neg_inf, not_a_num
# Out: (inf, -inf, nan)
```

`isinf`メソッドをして、またはののいずれかをテストできます。

```
math.isinf(pos_inf)
# Out: True

math.isinf(neg_inf)
# Out: True
```

のまたはのをしてテストすることができます。

```
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では、をチェックすることもできます

Python 3.x 3.2

```
math.isfinite(pos_inf)
# Out: False

math.isfinite(0.0)
# Out: True
```

は、とのにしてどおりにします。

```
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
```

しかし、が`float`としてできるよりきいをすると、になります。

```
pos_inf == sys.float_info.max * 1.0000001
# Out: True

neg_inf == -sys.float_info.max * 1.0000001
# Out: True
```

しかし、ゼロはのまたはなはのをもたらさず、むしろ`ZeroDivisionError`をさせます。

```
try:
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Division by zero")

# Out: Division by zero
```

のはのをもたらすだけで、には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は、それでさえ、にもしくはありません。 `isnan` メソッドを使ってテストすることができます

```
not_a_num == not_a_num
# Out: False

math.isnan(not_a_num)
Out: True
```

NaNはに「しくない」としてされますが、

```
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
```

NaNのはにNaNをえます。これには、-1によるがまれます。「のNaN」はありません。

```
5.0 * not_a_num
# Out: nan

float('-nan')
# Out: nan
```

Python 3.x 3.5

```
-math.nan
# Out: nan
```

NaNとのい `float` バージョンと Python 3.5の `math` ライブラリの中にはないがあります

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)
```

よりいのためのPow

コマンドラインから `timeit` モジュールをする

```
> 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
```

みみの**はしばしばですが、パフォーマンスがであるは、`math.pow`をしてください。しかし、がであっても、`pow`のりはすることにしてください。

```
> from math import pow
> pow(5,5)
3125.0
```

と`cmath`モジュール

`cmath`モジュールは`math`モジュールにありますが、`cmath`にしてなをしています。

まずに、は、ライブラリクラスによってされるのではなく、Pythonのであるです。したがって、のに`import cmath`を`import cmath`するはありません。

たちがしていることにしてください`j`または`J`とない`i`。

```
z = 1 + 3j
```

`j`はリテラルではなくのであるため、`1j`をするがあります。

```
1j * 1j
Out: (-1+0j)

1j ** 1j
# Out: (0.20787957635076193+0j)      # "i to the i" == math.e ** -(math.pi/2)
```

々はっている`real`と`imag`だけでなく、な`conjugate`

```
# 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
```

みみ`abs`と`complex`ものであり、インポートはありません。

```
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)
```

`complex`はをすることができますが、をつことはできません

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
```

```
# Exception: ValueError: complex() arg is a malformed string
```

しかし、ほとんどのでは、`sqrt` モジュールがです。

```
import cmath

cmath.sqrt(-1)
# Out: 1j
```

`sqrt` のとはとはなりません。な `math` では、ののはをさせます。

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

にするとからするがされています。

```
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)
```

なのはこののをえています、なのくのは、はまたはにった「カット」をとっています。のプラットフォームでは、IEEE 754でされている「きゼロ」がサポートされています。これにより、ブランチカットのでこれらののがされます。のは、Pythonのドキュメントです。

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
# Out: -3.141592653589793
```

`cmath` モジュールはまた、`math` モジュールからのなをつくのをします。

えて `sqrt`、なバージョンがある `exp`、`log`、`log10`、とその `sin`、`cos`、`tan`、`asin`、`acos`、`atan`、びとその `sinh`、`cosh`、`tanh`、`asinh`、`acosh`、`atanh`。しかし、`math.atan2` の2つのをとるの `math.atan2` のなはありません。

```
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
```

`pi` および `e` がされる。これらは `float` であり `complex` ではないことにしてください。

```
type(cmath.pi)
# Out: <class 'float'>
```

`cmath` モジュールは、`isinf` なバージョンと Python 3.2 の `isfinite` もします。「Infinity と NaN」をしてください。は、そのまたはのいずれかがである、とみなされます。

```
cmath.isinf(complex(float('inf'), 0.0))
# Out: True
```

に、`cmath` モジュールは `isnan` なバージョンをします。「Infinity と NaN」をしてください。は、そのまたはのいずれかが「ではない」、「ではない」とみなされます。

```
cmath.isnan(0.0, float('nan'))
# Out: True
```

くしない `cmath` の `math.inf` と `math.nan` Python の 3.5 から

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'
```

Python 3.5 では、`cmath` モジュールと `math` モジュールの `isclose` メソッドがあります。

Python 3.x 3.5

```
z = cmath.rect(*cmath.polar(1+1j))

z
# Out: (1.0000000000000002+1.0000000000000002j)

cmath.isclose(z, 1+1j)
# True
```

オンラインでモジュールをむ <https://riptutorial.com/ja/python/topic/230/モジュール>

181: の

き

がるためにデータをしてするときには、のフォーマットがにになります。 Pythonは、このトピックでされているさまざまなメソッドをしています。

- "{}".format(42)==> "42"
 - "{0}".format(42)==> "42"
 - "{0.2f}".format(42)==> "42.00"
 - "{0.0f}".format(42.1234)==> "42"
 - "{え}" format(no_answer = 41、 answer = 42==> "42"
 - "{え.2f}" no_answer = 41、 = 42==> "42.00"
 - "{[key]}" format{'key' 'value'}==> "value"
 - "{[1]}" フォーマット['zero'、 'one'、 'two']==> "one"
 - "{} = {}".format(42)==> "42 = 42"
 - ".join['stack'、 'overflow']==> "スタックオーバーフロー"
- [PyFormat.info](https://pyformat.info) をチェックして、それがどのようにするかについてののにでやかな/をてください。

Examples

フォーマットの

```
foo = 1
bar = 'bar'
baz = 3.14
```

`str.format` をしてをフォーマットすることができます。ブラケットペアは、がされたでにきえられます。

```
print('{}', '{} and {}'.format(foo, bar, baz))
# Out: "1, bar and 3.14"
```

のにインデックスをすることもできます。は `str.format` にされるのインデックスにします0ベース。

```
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
```

きもできます。

```
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."
```

`str.format`にすと、オブジェクトのをできます。

```
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"
```

キーもにできます。

```
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"
```

リストとタプルのインデックスにもじことがされます

```
my_list = ['zero', 'one', 'two']
print("2nd element is: {0[2]}".format(my_list)) # "0" is optional
# Out: "2nd element is: two"
```

`str.format`にえて、Pythonは、ののために、のまたは [PEP 3101](#)をとばれるモジュール
% `also`もしています。 `str.format`は`%str.format`あり、のをにできるなど、がします。

インデックスにえて、ののをめることもできます。これは、ルールに、コロンにしなければなら
ないです:。のについては、[ドキュメント](#)をしてください。のは、`:~^20` ^はをし、は20、でめ
ます

```
'{:~^20}'.format('centered')
# Out: '~~~~~centered~~~~~'
```

`format`は、のりしなど、%ではなをにします

```
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
```

`format`はなので、ののとしてうことができます

```
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

```

アライメントとパディング

Python 2.x 2.6

`format()` メソッドをすると、のことができます。 `:[fill_char][align_operator][width]` ここで、`align_operator` はのいずれかの `align_operator` です。

- `<` フィールドを `width` でえにします。
- `>` フィールドを `width` でえにします。
- `^` フィールドは `width` にセンタリングされ `width`。
- `=` のろにパディングをするようにしますのみ。

`fill_char` された、デフォルトはは、パディングにされるです。

```

'{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

'{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'

'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'

```

`ljust()`、`rjust()`、`center()`、`zfill()` をしてもじがられますが、これらはバージョン2.5ではされていません。

リテラル **f-string**

リテラルのは、[PEP 498](#) Python3.6でされました。リテラルのに `f` をして、のスコープのすべての `.format` ににすることができます。

```

>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'

```

これは、アラインメントやドットをむよりなフォーマットでもします。

```
>>> f'{foo:^7s}'  
'  bar  '
```

`f''`は、`bytes`は`b''`、`python2` `u''`は`unicode`のようなのをしません。はちにされ、のがわれる。

はネストすることもできます。

```
>>> price = 478.23  
>>> f'{f'${price:0.2f}':*>20s}'  
'*****$478.23'
```

`f-string`のは、からのにされます。これは、にがあるにのみです。

```
>>> 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]
```

`datetime`による

どのクラスでも、`__format__`メソッドをしてのをできます。これをにできるのPythonライブラリのは`datetime`で、`str.format`で`strftime`ようなコードをできます

```
>>> 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.'
```

[ドキュメント](#)には、フォーマッタのなリストがあります。

`Getitem`と`Getattr`をしたフォーマット

`__getitem__`をサポートするデータは、ネストされたをフォーマットできます。

```
person = {'first': 'Arthur', 'last': 'Dent'}  
'{p[first]} {p[last]}'.format(p=person)  
# 'Arthur Dent'
```

オブジェクトのには、`getattr()`をしてアクセスできます。

```
class Person(object):
```

```
first = 'Zaphod'
last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

フロートフォーマット

```
>>> '{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'
```

ののためのじホールド

```
>>> '{:.3f}'.format(42.12345)
'42.123'

>>> '{answer:.3f}'.format(answer=42.12345)
'42.123'
```

は、またはパーセントですることでもあります。

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'

>>> '{0:.0%}'.format(42.12345)
'4212%'
```

{0} と {name} をみわせることもできます。これは、すべてのを1つののであらかじめされたにめるにです。

```
>>> 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'
```

の

.format() メソッドは、のようなるフォーマットのをできます。

```
>>> '{:c}'.format(65)    # Unicode character
'A'

>>> '{:d}'.format(0x0a)  # base 10
'10'

>>> '{:n}'.format(0x0a)  # base 10 using current locale for separators
'10'
```

をなるベース**16**、**8**、**2**にフォーマットする

```
>>> '{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'
```

をして、RGBフロートタプルをカラーの16にします。

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{0:02X}{0:02X}{0:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'FF6600'
```

だけをすることができます

```
>>> '{: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'
```

クラスのカスタム

のすべては`str.format`メソッドと`format`にされます。のテキストでは、2つはじです。

`format`にされるすべてののにして、Pythonはそののにして`__format__`メソッドをします。したがって、のカスタムクラスでの`__format__`メソッドをして、`format`がクラスとそののをどのようにおよびフォーマットするかをすることができます。

これは、よりなる`__str__`のように、`__format__`アカウントにわせ、フィールドなどをむフォーマットを、とさえであれば、のをし、ののすることができます。 [1](#)

```
object.__format__(self, format_spec)
```

例えば

```
# 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.
```

カスタムクラスにカスタム`__format__`メソッドがなく、クラスのインスタンスが`format`にされた、**Python2**は`__str__`メソッドまたは`__repr__`メソッドのりをしてをするかをします。デフォルトの`repr`がされます、フォーマットするには`s`フォーマットをするがあります。**Python3**では、カスタムクラスを`format`にすには、カスタムクラスで`__format__`メソッドをするがあります。

ネストされた

のでは、されたのやなど、のパラメータをできます。

```
>>> '{:.>10}'.format('foo')
'.....foo'
```

それらはまた、パラメータとしてすることができる`format`よりネストによって`{}`を`{}`

```
>>> '{:.>{}}'.format('foo', 10)
'.....foo'
'{:({}){}}'.format('foo', '*', '^', 15)
'*****foo*****'
```

のでは、`'{:({}){}}'`が`'{:.*^15}'`されていますの、`foo`はそうようにフォーマットされます。

これは、のデータをするに、にパラメータがわからないにちます。

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
bbbbbbb
      ccc
```

のパディングとランケート、みわせ

を3のにしたいとします。

{と}はそれらをエスケープします。

```
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))
```

```
pad
{:3}           :1  :

truncate
{:~.3}         :555:

combined
{:>3.3}        : 1:
{:3.3}         :1  :
{:3.3}         :333:
{:3.3}         :555:
```

きブレースホルダ

には、のためにキーワードをしてされたきブレースホルダがまれformat。

をうPython 2.x

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

をうPython 3.2

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

`str.format_map`をすると、をにしなくてもをできます。また、`data`のクラスカスタムタイプかもしれないが、しくめまれた`dict`わりにされます。

なし

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
'Hodor Hodor!'
```

オンラインでのをむ <https://riptutorial.com/ja/python/topic/1019/>の

182: またはリストからのCSVへのきみ

き

.csvファイルへのきみは、のファイルへのきみとほとんどいはなく、かなりです。は、なので、にするもでなアプローチをカバーします。

パラメーター

パラメータ	
オープン " / path /" 、 "mode"	CSVファイルへのパスをする
オープンパス、 "モード"	ファイルをくためのモードをするみみ、きみなど
csv.writer ファイル、り	かれたCSVファイルをここにします
csv.writerファイル、り= "	りまたはパターンをする

```
open( path, "wb")
```

"wb" - きみモード。

たちがした"wb"のbパラメータは、バイナリモードでくにのみです。バイナリモードは、Windowsなどのオペレーティングシステムでのみです。

```
csv.writer ( csv_file, delimiter=',' )
```

ここでは、しているりは、ある,々はデータのセルは、それぞれ、およびをむようにしたいので、。たちのリストは,あまりにもかれていますので、たちのがはるかにです。

Examples

なきみの

```
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)

```

CSVファイルにをえる

```

def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "\n")

```

オンラインでまたはリストからのCSVへのきみをむ <https://riptutorial.com/ja/python/topic/10862/>
 またはリストからのcsvへのきみ

183: メソッド

- `str.capitalize` - > str
- `str.casefold` - > str [Python> 3.3のみ]
- `str.centerwidth` [、 fillchar] - > str
- `str.countsub` [、 start [、 end]] - > int
- `str.decodeencoding` = "utf-8" [、 errors] - > unicode [Python 2.xのみ]
- `str.encodeencoding` = "utf-8"、 errors = "strict" - > バイト
- `str.endswith`[、 start [、 end]] - > bool
- `str.expandtabstabsz` = 8 - > str
- `str.findsub` [、 start [、 end]] - > int
- `str.format`* args、 ** kwargs - > str
- `str.format_map` マッピング - > str
- `str.indexsub` [、 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.joiniterable` - > str
- `str.ljustwidth` [、 fillchar] - > str
- `str.lower` - > str
- `str.lstrip`[chars] - > str
- static `str.maketransx` [、 y [、 z]]
- `str.partitionsep` - > head、 sep、 tail
- `str.replaceold`、 new [、 count] - > str
- `str.rfindsub` [、 start [、 end]] - > int
- `str.rindexsub` [、 start [、 end]] - > int
- `str.rjustwidth` [、 fillchar] - > str
- `str.rpartitionsep` - > head、 sep、 tail
- `str.rsplitsep` = None、 maxsplit = -1 - > のリスト
- `str.rstrip`[chars] - > str
- `str.splitsep` = None、 maxsplit = -1 - > のリスト
- `str.splitlines`[keepends] - > のリスト
- `str.startswith`[、 start [、 end]] - >
- `str.strip`[chars] - > str
- `str.swapcase` - > str

- `str.title` - > `str`
- `str.translatetable` - > `str`
- `str.upper` - > `str`
- `str.zfillwidth` - > `str`

Stringオブジェクトはです。つまり、リストのようにすることはできません。このため、`str`メソッドは、にメソッドびしのをむしい `str` オブジェクトをします。

Examples

のとの

Pythonのは、のにするくのをします。これらには、

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

UnicodePython 3のデフォルトでは、これらは11マッピングでもでもありません。これらののほとんどは、ではなく、のためのものです。

Python 3.x 3.3

`str.casefold()`

`str.casefold`は、とをしないにしたのをします。これは`str.lower`よりもで、すでにのをしたり、のさをばしたりすることがあります。ではありません。

```
"XBΞ".casefold()
# 'xssσ'

"XBΞ".lower()
# 'xβς'
```

ケースフォールディングのもとでわれるは、WebサイトのCaseFolding.txtファイルのUnicode Consortiumによってされます。

`str.upper()`

`str.upper`はのすべてのをり、それをにします。

```
"This is a 'string'.".upper()
# "THIS IS A 'STRING'."
```

```
str.lower()
```

`str.lower`はです。のすべてのをり、にします。

```
"This IS a 'string'".lower()
# "this is a 'string'."
```

```
str.capitalize()
```

`str.capitalize`はのをします。つまり、のはにし、りのはくします。

```
"this IS A 'String'".capitalize() # Capitalizes the first character and lowercases all others
# "This is a 'string'."
```

```
str.title()
```

`str.title`は、のタイトルに`str.title`バージョンをします。つまり、のにあるすべてののはになり、それはすべてになります。

```
"this IS a 'String'".title()
# "This IS A 'String'"
```

```
str.swapcase()
```

`str.swapcase`は、すべてのがにスワップされ、すべてがにスワップされるしいオブジェクトをします。

```
"this iS A STRiNg".swapcase() #Swaps case of each character
# "THIS Is a strIng"
```

`str` クラスメソッドとしての

これらのメソッドは、オブジェクトのようにまたは`str`クラスのクラスメソッド `str.upper`へのなびしなどでびすことが`str.upper`。

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

これは、これらのメソッドの1つをくのにに`map`でするときにもです。

```
map(str.upper, ["These", "are", "some", "'strings'"])
# ['THESE', 'ARE', 'SOME', "'STRINGS'"]
```

りにづいてをのリストにする

```
str.split(sep=None, maxsplit=-1)
```

`str.split`はをけり、のののリストをします。は、`sep`がされているかされているかによってなります。

`sep`がされていない、または`None`がされているは、があるであればがわれます。しかし、とのはされ、のしたは1つのとじようにわれます。

```
>>> "This is a sentence.".split()
['This', 'is', 'a', 'sentence.']

>>> " This is    a sentence. ".split()
['This', 'is', 'a', 'sentence.']

>>> "          ".split()
[]
```

`sep`パラメータは、りをするためにできます。のは、りがあるところでされ、りはされます。のすりは、のオカレンスとじようにわれるのではなく、むしろのがされます。

```
>>> "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.']
```

デフォルトでは、デリミタがするたびにされますが、`maxsplit`パラメータはするをします。デフォルト-1はをします

```
>>> "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`「スプリット」とはなる`str.split`とき「スプリット」`maxsplit`されています。は、で

はなくのからされます。

```
>>> "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', '.']
```

Pythonはされるのをしますが、のほとんどのプログラミングはされるのをします。これは、コードをまたはするときにくがあります。

あるのすべてののをのにきえる

Pythonの`str`には、あるサブのを、されたののサブにきえるメソッドもあります。よりのしいには、`re.sub`

```
str.replace(old, new[, count])
```

`str.replace`は、`new`サブできえられる`old`サブをむ`old`と`new` 2つのをとります。オプションの`count`は、きえの`count`します。

たとえば、ので、`'foo'`を`'spam'`に`str.replace`、`old = 'foo'`と`new = 'spam'` `str.replace`をびすことができます。

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
'Make sure to spam your sentence.'
```

されたに`old`とするのがまれている、すべてののは`new`されたにきえられます。

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
'It can spam multiple examples of spam if you want.'
```

もちろん、`count`をしないうこの、`count`オカレンスはきえられます

```
>>> """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`とf-stringsをにフォーマットする

Pythonは、バージョン2.6でされた`str.format`とバージョン3.6でされたf-stringsによって、のとをします。

のがえられた

```
i = 10
```

```
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

のはすべてです

```
"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}"
```

までに、PythonはフォーマットのCスタイルのもサポートしています。のとは `str.format` ですが、
、の、というから、 `str.format` バージョンがされます。

```
"%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)
```

`str.format` にするは、のにをらすためにを `str.format` こともできます。たとえば、はです。

```
"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")
```

のpythonのドキュメントは、いつものように、にしていますが、 pyformat.info にはなをむらしい
があります。

さらに、`{ }` と `{}_` はでエスケープできます。

```
"{'a': 5, 'b': 6}"
>>> "{{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)
>>> f"{{{'a'}}: {5}, {'b'}}: {6}}"
```

は、「 **フォーマット** 」をしてください。 `str.format()` は [PEP 3101](https://peps.python.org/pep-3101/) でされ、f-stringsは [PEP 498](https://peps.python.org/pep-498/) で

されました。

にがれるをえる

の`str.count` サブのをカウントする1つのがあります。

```
str.count(sub[, start[, end]])
```

`str.count` は、のの `sub` していないオカレンスのをす `int` をします。オプションの `start` と `end` はの `start` と `end` します。デフォルトで `start = 0` と `end = len(str)` は、がされることをし `end = len(str)` 。

```
>>> s = "She sells seashells by the seashore."
>>> s.count("sh")
2
>>> s.count("se")
3
>>> s.count("sea")
2
>>> s.count("seashells")
1
```

のをすることで `start`、`end` 々はよりなをることができるのであれば、例えば、カウント `start` にしい ¹³ へのびし

```
>>> s.count("sea", start)
1
```

のものとして。

```
>>> t = s[start:]
>>> t.count("sea")
1
```

のをテストする

Pythonでえられたのめとわりをテストするために、`str.startswith()` と `str.endswith()` メソッドをうことができます。

```
str.startswith(prefix[, start[, end]])
```

がするように、`str.startswith` は、されたが `prefix` のされたでまるかどうかをテストするためにされ `prefix` 。

```
>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
```



```
>>> s.startswith("thi")
False
```

オプションの`start`と`end`は、テストの範囲を指定します。ここでは、`2`することによって、`s[2]`からされます。

```
>>> s.startswith("is", 2)
True
```

これは、`s[2] == 'i'`と`s[3] == 's'` `True` `s[3] == 's'` ます。

`tuple`をして、のセットでまるかどうかをべることもできます

```
>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False
```

`str.endswith(prefix[, start[, end]])`

`str.endswith`は`str.endswith`とまったく同じ`str.startswith`が、のいはをし、はしないことです。たとえば、がしてするかどうかをテストするには、のようになります。

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!!')
False
```

`startswith`と`startswith`に、のをシーケンスとしてできます。

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

`tuple`をして、のセットでわるかどうかをべることもできます

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

のをテストする

Pythonの`str`には、のをするためにできるいくつかのメソッドもあります。これらは`str.isalpha`、`str.isdigit`、`str.isalnum`、`str.isspace`です。と`str.isupper`、`str.islower`、`str.islower`、および`str.istitle`でテストできます。

`str.isalpha`

`str.isalpha` はをりません。また、されたのすべてののがアルファベットののは`True`をします。

```
>>> "Hello World".isalpha() # contains a space
False
>>> "Hello2World".isalpha() # contains a number
False
>>> "HelloWorld!".isalpha() # contains punctuation
False
>>> "HelloWorld".isalpha()
True
```

の、のは`"".isalpha()`すると`False`にされます。

`str.isupper` 、 `str.islower` 、 `str.istitle`

これらのメソッドは、されたのをテストします。

`str.isupper` は、されたのすべてののがのは`True`をし、そうでないは`False`をすメソッドです。

```
>>> "HeLLO WORLD".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> "".isupper()
False
```

に、`str.islower` は、されたのすべてののがであれば`True`をし、それのは`False`をすメソッドです。

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` は、されたがタイトルにまれるは`True`します。つまり、すべてののはでまり、そのにかきます。

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

`str.isdecimal` 、 `str.isdigit` 、 `str.isnumeric`

`str.isdecimal` は、が10のシーケンスであるかどうかをします。これは10をすのにしています。

`str.isdigit` は、きなどのをすのにしていながまれます。

`str.isnumeric` は、09ののなど、でなくてものがまれます。

	<code>isdecimal</code>	<code>isdigit</code>	<code>isnumeric</code>
12345	True	True	True
12345	True	True	True
12345	False	True	True
12345	False	False	True
Five	False	False	False

BytestringsPython 3の`bytes`、Python 2の`str` は、`isdigit`のみをサポートし、なASCIIのチェックのみをいます。

`str.isalpha` に、はFalseされます。

`str.isalnum`

これは`str.isalpha`と`str.isnumeric`みわせです。には、されたのすべてののがである、つまりまたはでされているにTrueとされます。

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
>>> "Hello World".isalnum() # contains whitespace
False
```

`str.isspace`

にのみがまれている、TrueされTrue。

```
>>> "\t\r\n".isspace()
True
>>> " ".isspace()
True
```

にはが""にえることもありますが、がかがまったくまれていないかどうかはわかりません

```
>>> "".isspace()
False
```

このケースをカバーするには、テストがです

```
>>> my_str = ''
>>> my_str.isspace()
False
>>> my_str.isspace() or not my_str
True
```

しかし、がであるかしがまかれていないかをテストするのは、`strip`をすることですなしでとのをすべてします

```
>>> not my_str.strip()
True
```

str.translateのの

Pythonはstrのtranslateメソッドをサポートしていstrこれは、テーブルきえにされると、プロセスでされるべきをすることをにします。

```
str.translate(table[, deletechars])
```

パラメータ	
table	あるからのへのマッピングをするルックアップテーブルです。
deletechars	からされるのリスト。

maketrans str.maketransはPython 3でとstring.maketransのPython 2でをすると、テーブルをすることができます。

```
>>> translation_table = str.maketrans("aeiou", "12345")
>>> my_string = "This is a string!"
>>> translated = my_string.translate(translation_table)
'Th3s 3s 1 str3ng!'
```

translateメソッドは、ののされたコピーであるをします。

をするがあるのみ、tableをNoneできます。

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
'ths syntx s vry sfl'
```

からなIをりく

str.strip、str.rstrip、およびstr.lstrip、のとのをりく3つのメソッドがされています。3つのメソッドはすべてじシグネチャをち、3つすべてがながされたしいオブジェクトをします。

```
str.strip([chars])
```

`str.strip()`は、されたにし、`chars`まれるまたはのをします。`chars`がされていない、または`None`、すべてのがデフォルトでされます。えは

```
>>> "    a line with leading and trailing space    ".strip()
'a line with leading and trailing space'
```

`chars`がされた、そのにまれるすべてののがからされ、されます。えは

```
>>> ">>> a Python prompt".strip('> ') # strips '>' character and space character
'a Python prompt'
```

`str.rstrip([chars])` および `str.lstrip([chars])`

これらのメソッドは`str.strip()`とのセマンティクスとを`str.strip()`ますが、そのいはのにあります。`str.rstrip()`はのからし、`str.lstrip()`はのからします。

たとえば、`str.rstrip`をし`str.rstrip`。

```
>>> "    spacious string    ".rstrip()
'    spacious string'
```

、`str.lstrip`をして

```
>>> "    spacious string    ".rstrip()
'spacious string    '
```

とをしない

とをしないでするのはなことですが、そうではありません。このセクションでは、UnicodeのみをしますPython 3のデフォルト。Python 2はPython 3にべてながあるかもしれないことにしてください。のUnicodeははるかにです。

に、ユニコードでとをするがではないことにしてください。"ß"のような`text.lower() != text.upper().lower()`テキストがあります

```
>>> "ß".lower()
'ß'

>>> "ß".upper().lower()
'ss'
```

しかし、あなたは"BUSSE"と"Buße" "BUSSE"したいとしましょう。"BUSSE"、あなたはおそらく"BUSSE"と"BUßE"にしたいとかもしれません。それはよりしいです。`casefold`をすることをおめします。

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.
"""
```

lower しないでください。 casefold をできないは、 .upper().lower() するとちますただし、ののみ。

に、アクセントをするがあります。あなたのフォントレンダラーがいなら、おそらく "ê" == "ê " とうかもしれませんが、そうではありません

```
>>> "ê" == "ê "
False
```

これは、には

```
>>> 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']
```

これをするもなは unicodedata.normalize です。 **NFKD** のをしたいとうかもしれませんが、ドキュメントをチェックしてください。それから、

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê ")
True
```

これをけるために、ここではでします

```
import unicodedata

def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

のリストを1つのにする

をセパレータとしてして、 join() メソッドをしてのリストを1つのにまとめることができます。たとえば、リストのがスペースでられたをできます。

```
>>> " ".join(["once", "upon", "a", "time"])
```

```
"once upon a time"
```

のでは、を3つのハイフンでります。

```
>>> "---".join(["once", "upon", "a", "time"])
"once---upon---a---time"
```

モジュールのな

Pythonのstringモジュールは、にするのためのをします。それらをするには、stringモジュールをインポートします。

```
>>> import string
```

string.ascii_letters

ascii_lowercaseとascii_uppercase

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

string.ascii_lowercase

すべてののASCIIがまれます。

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

string.ascii_uppercase

すべてののASCIIがまれます。

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

string.digits

すべての10のがまれます。

```
>>> string.digits
'0123456789'
```

string.hexdigits

すべての16をみます

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

`string.octaldigits`

すべての8のがまれます。

```
>>> string.octaldigits
'01234567'
```

`string.punctuation`

☞ロケールでとみなされるすべてのがまれます。

```
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

`string.whitespace`

とみなされるすべてのASCIIをみます

```
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

スクリプトモードでは、`print(string.whitespace)`はのをし、`str`をしてにされたをします。

`string.printable`

とみなされるすべてのをみます。 `string.digits`、`string.ascii_letters`、`string.punctuation`、および`string.whitespace`。

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

をする

をり、のでイテレータをすみみの`reversed()`をして、をにすることができます。

```
>>> reversed('hello')
<reversed object at 0x0000000000000000>
>>> [char for char in reversed('hello')]
```



```
['o', 'l', 'l', 'e', 'h']
```

`reversed()` のびしでラップすることができます `''.join()` をるためにイテレータからを。

```
>>> ''.join(reversed('hello'))
'olleh'
```

されていないPythonユーザーは `reversed()` をするがみやすくなりますが、`-1` ステップで `スライス` をするがでになります。ここでは、としてしようします

```
>>> def reversed_string(main_string):
...     return main_string[::-1]
...
>>> reversed_string('hello')
'olleh'
```

のびえ

Pythonはをするためのをし、さまざまなをよりにさせるためのテキストパディングをにします。

は `str.ljust` と `str.rjust` です

```
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` と `rjust` はにています。どちらも `width` パラメータとオプションの `fillchar` パラメータを `fillchar` ます。これらのでされたは、なくともにされた `width` パラメータとじさです。の `width` `already` よりもい、はりてられません。 `fillchar` は、デフォルトで、になります、マルチキャラクターではなく、でなければなりません。

`ljust` は、 `fillchar` びされたのわりを `width` になるまで `fillchar` ます。 `rjust` は、のでのを `rjust` ます。したがって、これらののの `l` と `r` は、 `fillchar` ではなくのがにされているをします。

str または **bytes** データと **unicode** のの

ファイルおよびネットワークメッセージのは、されたをすることがあります。なのために、しばし

ばユニコードにするがあります。

Python 2では、strデータをUnicodeにするがあります。デフォルト、`' '`、`''' '''`などはASCIIで、ASCIIのはエスケープとしてされます。Unicodeは`u' '` または `u''' '''` などです。

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
```

Python 3では、バイトのバイトリテラルとばれるをUnicodeにするがあります。デフォルトでは、Unicodeで、バイトリテラルは、のようにするがあります`b' '`、`b''' '''` されますなどのバイトリテラルのTrueの`isinstance(some_val, bytes)`とすると、`some_val` エンコードされるかもしれないであることをバイト。

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.
```

がまれています

Pythonでは、にされたがまれているかどうかをチェックするのがにです。ただ、`in`オペレータ

```
>>> "foo" in "foo.baz.bar"
True
```

のをテストするとに`True`。

```
>>> "" in "test"
True
```

オンラインでメソッドをむ <https://riptutorial.com/ja/python/topic/278/メソッド>

184: の

Examples

2つの

```
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
```

を**datetime**オブジェクトにする

Cの**フォーマットコード**をします。

```
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)
```

に**datetime**オブジェクトをする

Cの**フォーマットコード**をします。

```
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
```

オンラインでのをむ <https://riptutorial.com/ja/python/topic/7284/>の

185:

Pythonは、とを、、、するためのみみメソッドとライブラリのをします。

Examples

をタイムゾーンの`datetime`オブジェクトにする

を`datetime`オブジェクトにすると、Python 3.2では`%z`をサポートしています。

`+HHMM`または`-HHMM`のUTCオフセットオブジェクトがナイーブな。

Python 3.x 3.2

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

Pythonのバージョンでは、`dateutil`のようなライブラリをうことができます。これは、タイムゾーンきのをして`datetime`オブジェクトにすることをくいます。

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

`dt`は、のを`datetime`オブジェクトになりました。

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

な

はしてしません。あなたは、ののをつけるか、またはがになるかをするがあるのがです。これは`timedelta`オブジェクトをして`timedelta`できます

```
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)
```

これにより、のようながられます。

```
Today: 2016-04-15
Yesterday: 2016-04-14
Tomorrow: 2016-04-16
Difference between tomorrow and yesterday: 2 days, 0:00:00
```

なdatetimeオブジェクトの

datetimeモジュールには、、、の3つのなオブジェクトがまれています。

```
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)
```

これらのオブジェクトのは、じデータでのみサポートされ、なるのインスタンスでなをすると、TypeErrorがします。

```
# 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)
```

をする

には、からまでのあるのにわたってりしたいとうことがあります。あなたはdatetimeライブラリとtimedeltaオブジェクトをしてそれをうことができます

```
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)
```

するもの

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

いタイムゾーンをつをタイムゾーンの**datetime**オブジェクトにする

のように`dateutil`ライブラリをして**タイムゾーンのタイムスタンプ**をすると、された「い」タイムゾーンでタイムスタンプをすることもできます。

にあいまいないのまたはでフォーマットされたのたとえば、キューバなどのCSTなど - は**こちら**、データベースではずしもできないタイムゾーンと`tzinfo`オブジェクトののマッピングをすることができます。

```
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)
```

これをした

```
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'))
```

これで`pytz`タイムゾーンをすると、にローカライズされないことにしてください。

```
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})
```

これは`pytz`タイムゾーンを`datetime`ににするだけです

```
dt.tzinfo # Will be in Local Mean Time!  
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

このメソッドをしているのは、した、おそらくのなを`localize`するがあります。

```
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>
```

タイムゾーンをするの

デフォルトでは、すべての`datetime`オブジェクトは`naive`です。タイムゾーンをさせるには、UTCのオフセットとタイムゾーンののをのとしてする`tzinfo`オブジェクトをするがあります。

オフセットタイムゾーン

Python 3.2では、UTCからのオフセットである`timezone`、`datetime`モジュールは`timedelta`とオプションの`name`パラメータを`tzinfo`なである`timezone`クラスをします。

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'
```

3.2よりのバージョンのPythonでは、`dateutil`などのサードパーティのライブラリをするがあります。`dateutil`クラス、`tzoffset` バージョン2.5.3のようなのを、`dateutil.tz.tzoffset(tzname, offset)`、`offset`でされます。

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, 0, tzinfo=JST)  
print(dt)  
# 2015-01-01 12:00:00+09:00  
print(dt.tzname())
```



```
# 'JST'
```

のあるゾーン

のゾーンの、Pythonライブラリはクラスをしないため、サードパーティライブラリをするがあります。 [pytz](#)と[dateutil](#)は、タイムゾーンクラスをするなライブラリです。

なにえて、 [dateutil](#)はをするタイムゾーンクラスをします [tzモジュールのドキュメント](#)を。

`tz.gettz()` メソッドをしてタイムゾーンオブジェクトをし、それを `datetime` コンストラクタにすことができます。

```
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
```

バージョン2.5.3では、 [dateutil](#)はあいまいなをし、にのにデフォルトされます。え、2015-11-01 1:30 EDT-4す [dateutil](#) タイムゾーンをつオブジェクトをするはありません。これは、のであるためです。

[pytz](#)をする、すべてのエッジケースがにされますが、 [pytz](#) タイムゾーンは、コンストラクタをしてタイムゾーンにされるべきではありません。わりに、 [pytz](#) タイムゾーンをタイムゾーンの `localize` メソッドをしてするがあります。

```
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
```

[pytz](#) するタイムゾーンでをする、がなはUTCでをするか、にして `normalize()` をびすがあります。

```
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
```

ファジイテキストからdatetimeをする

のであるとされないのがされる「ファジー」モードの `dateutil` パーサーをして、テキストからをすることはです。

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

`dt` は `datetime` オブジェクトになり、 `datetime.datetime(2047, 1, 1, 8, 21)` されます。

タイムゾーンのりえ

タイムゾーンをりえるには、タイムゾーンをするオブジェクトがです。

```
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.
```

のライブラリでのISO 8601タイムスタンプをする

Pythonは、ISO 8601のタイムスタンプのをしています。 `strftime`、どのであるかをにるがあります。なとして、 `datetime` のはISO 8601のタイムスタンプで、スペースはりと6のになります。

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))
# '2016-07-22 09:25:59.555555'
```

が0のははされません

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

しかし、これらの2 つのは、 `strftime` ためになるをとします。さらに、 `strftime` ' does not support at all parsing minute timezones that have a ° in it, thus `20167220925+ 0300 59` を can be parsed, but the standard format `201672292559` を `+0300` はできません。

`iso8601` とばれる `ファイル` ライブラリがあり、ISO 8601のタイムスタンプとその `ファイル` のみをし

くします。

とタイムゾーンをサポートし、`T`セパレータはすべてのでサポートされています

```
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' ...>)
```

`timezone`がされていない、`iso8601.parse_date`デフォルトはUTCになります。デフォルトゾーンは`default_zone`キーワードでできます。に、これがデフォルトのわりに`None`である、なタイムゾーンをたないタイムスタンプは、わりに`naive datetimes`としてされます。

```
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>)
```

タイムスタンプを`datetime`にする

`datetime`モジュールは、POSIX `timestamp`をITC `datetime`オブジェクトにできます。

エポックは197011のです。

```
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)
```

からにをしく

`calendar`モジュールの

```
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)
```

`dateutils`モジュールの

```
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)
```

の

timedelta モジュールはのをするのにです

```
from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23) # datetime.datetime(2016, 05, 23, 0, 0, 0)
```

しいdatetime オブジェクトをする、のはオプションです。

```
delta = now-then
```

delta はtimedeltatimedelta

```
print(delta.days)
# 60
print(delta.seconds)
# 40826
```

のnとnにするには、のものをします。

n

```
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)

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)
```

ISO 8601 タイムスタンプをする

タイムゾーンなし、マイクロ

```
from datetime import datetime

datetime.now().isoformat()
# Out: '2016-07-31T23:08:20.886783'
```

タイムゾーンでは、マイクロ

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# Out: '2016-07-31T23:09:43.535074-07:00'
```

マイクロなしのタイムゾーン

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Out: '2016-07-31T23:10:30-07:00'
```

ISO 8601については、 [ISO 8601](#) をしてください。

オンラインでをむ <https://riptutorial.com/ja/python/topic/484/>

186:

き

if、elif、elseなどのキーワードをむは、ブールにじてなるアクションをするをPythonプログラムにしますTrueまたはFalse。このセクションでは、Pythonの、ブール、およびのについてします。

- <> if <き> else <>

Examples

if、elif、else

Pythonでは、あなたはしてのシリーズをすることができるifの1、のためelifアップまでのりのため、にオプション else、のでされないもののために。

```
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.")
```

Number is bigger than 2

elifわりにelse ifをするとエラーがし、されません。

または「」

は、インラインにされます。にみれるシンプルでなにです。

- のは、のくのC、Ruby、Javaなどとはなり、Pythonの「くべき」にれていないがそれをするがになることがあるにバグにつながるがあります。
- それはののれをにえ、にをえるにするので、「いにくい」とじるもいます。

```
n = 5

"Greater than 2" if n > 2 else "Smaller than or equal to 2"
# Out: 'Greater than 2'
```

こののは、でみまれたになります。がTrueのは、のに、それのはにされます。

テナントは、のようにネストすることもできます。

```
n = 5
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

また、 [ラムダに](#)をめるもします。

if

```
if condition:
    body
```

if ステートメントは条件をチェックします。 True とされると、 if の後の行が実行されます。 False とされた場合は、if の後の行をスキップします。

```
if True:
    print "It is true!"
>> It is true!

if False:
    print "This won't get printed.."
```

if には条件を指定できます。

```
if 2 + 2 == 4:
    print "I know math!"
>> I know math!
```

Else ステートメント

```
if condition:
    body
else:
    body
```

if が False にされたときに、else はその後の行を実行します。

```
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!
```

ブール

ブールは、にすることにえて、 `True` または `False`、とされたをす `True` または `False`。 `if-else` テストがなロジックをするのはPythonのです。

そしてオペレータ

`and` はすべてのをし、すべてのが `True` されたはのをします。それのは、 `False` とされるのをします。

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

または

`or` オペレータは、からにをし、にのをします `True` もしない、またはの `True`。

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

レイジー

このアプローチをするは、がであることにしてください。をするためにするのはいはされません。えば

```
>>> def print_me():
    print('I am here!')
>>> 0 and print_me()
0
```

のでは、 `print_me` はしてされません。なぜなら、Pythonは `0 False` にするとが `False` できるからで

す。プログラムロジックをするために`print_me`をするがあるは、このことをえておいてください。

ののテスト

のをチェックするのよくあるいは、ロジックをとってすることです。

このでは、2つのがそれぞれ2よりきいかどうかをチェックしようとしています。これは、`-if if` `(a) and (b > 2)`としてされます。 `bool(a)`は、 `a`が0でないに`True`されるため、しないになります。

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')

yes
```

は々にするがあります。

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')

no
```

がののいずれかであるかどうかをチェックするにも、のいがあります。こののステートメントは、 `-if (a == 3) or (4) or (6)`としてされます。これは、 `bool(4)`と`bool(6)`それぞれ`True`されるため、しないになり`True`

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
...     print('no')

yes
```

、は々にわなければならない

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')

no
```

inをするのがなです。

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')

no
```

の

のは、ブールにされたときにFalseとされるため、Falseとみなされます。

- し
-
- 0、または、えげろにするの、0L、0.0、0j
- のシーケンス ''、""、()、[]
- のマッピング {}
- __bool__または__len__メソッドが0またはFalseすユーザー

PythonののすべてののはTrueされTrue。

よくあるいは、いがなになるのをすのをにチェックすることです。たとえば、if foo() がよりではif foo() is None

cmpをして2つのオブジェクトのをする

Python 2にはcmpがまれています。このをすると、あるオブジェクトがのオブジェクトよりもさいかしいかきいかをできます。これは、これらの3つのオプションのいずれかについてリストからをするためにできます。

x > y 'greater than'は'greater than' 'less than'、 x < y 'less than'は'less than' 'equal'、 x == y 'equal'は'equal'をするがあるとします。

```
['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)はのをします

x < y	-1
x == y	0

```
x > y    1
```

これは、あなたができるのPython 3でされ`cmp_to_key(func)`にしてヘルパー`functools`キーををするためのPython 3でを。

リストによる

Pythonでは、リストをハックしてをすることができます。

えば、

```
[value_false, value_true][<conditional-test>]
```

```
>> n = 16
>> print [10, 20][n <= 15]
10
```

ここで`n<=15`は`False`しますこれはPythonでは0になります。だからPythonがしているものはのとおります。

```
[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

みみの`__cmp__`メソッドは3つのなをしました0,1、-1、`cmpx、y`がされた0のオブジェクトがじ`1x>y -1x<y`

これは、リストとともに、リストののつまり、インデックス0、2のつまりインデックス1、のつまりインデックス-1のをすためにできます。たちにこのタイプのをえます

```
[value_equals, value_greater, value_less][<conditional-test>]
```

に、のすべてので、Pythonはのブランチをするにのブランチをします。したブランチだけををするには

```
[lambda: value_false, lambda: value_true][<test>]()
```

に`()`をすると、にラムダがびされ/されることがされます。したがって、したのみをします。

```
count = [lambda:0, lambda:N+1][count==N]()
```

オブジェクトが`None`であるかどうかをテストし、それをりて

オブジェクトが与えられていないことを `is None`、そのオブジェクトにかゝりてることがよくあります。私たちは `aDate` を `is None` とし `aDate`。

これをうもな `is None` テストをすること `is None`。

```
if aDate is None:
    aDate=datetime.date.today()
```

Pythonicという `is None`、`== None` ではなく `is None` あることにしてください。

しかし、これは、`not None` がブールで `True` とされ `not None` というをすることによってわずかにできます。のコードはです

```
if not aDate:
    aDate=datetime.date.today()
```

しかしもっとPythonのがあります。のコードもじです

```
aDate=aDate or datetime.date.today()
```

これはをいいます。 `aDate` がされて `not None` で `not None`、ネット `not None` でそれにりてられます。 `is None`、`datetime.date.today()` は `aDate` にりてられ `aDate`。

オンラインでをむ <https://riptutorial.com/ja/python/topic/1111/>

187:

Examples

イメージファイルを読み込む

```
from PIL import Image

im = Image.open("Image.bmp")
```

ファイルをJPEGにする

```
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)
```

オンラインで読む <https://riptutorial.com/ja/python/topic/6841/>

き

Shelveは、オブジェクトをファイルにするためにされるPythonモジュールです。shelveモジュールは、ディクショナリのようなAPIをして、ピクルすることができるのPythonオブジェクトのストレージをします。シエルブモジュールは、リレーショナルデータベースがにされている、Pythonオブジェクトのシンプルなストレージオプションとしてできます。シエルフは、とに、キーでアクセスされます。は、pickdbによって、anydbmによっておよびされるデータベースにきまれます。

がにじられることにらないでください。それをもはやとしないときはにclose()びすか、context managerとしてshelve.open()をいます

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

ので、shelveモジュールによってえられpickle、できないソースからのシエルフをロードするためにではありません。pickleとに、シエルフをみむとのコードをできます。

1。するデータベースパッケージdbm.ndbmまたはdbm.gnuなどの、なインターフェイスによってなります。したがって、dbmをしてデータベースをくのはではありません。データベースは、ながらされているはdbmのにいます。つまり、データベースにされているオブジェクトはピクルされたはかなりさくなくればならず、まれにキーのによりデータベースがをする

2.シエルフモジュールは、シエルフオブジェクトにするみきアクセスをサポートしていません。のみみアクセスはです。きみのがいているは、のプログラムでみりまたはきみにくはありません。これをするにはUnixファイルロックをすることができますが、これはUnixバージョンによってなり、されるデータベースにするがです。

Examples

shelveのサンプルコード

オブジェクトをシエルフするには、まずモジュールをインポートし、のようにオブジェクトをりてます。

```
import shelve
database = shelve.open(filename.suffix)
object = Object()
database['key'] = object
```

インタフェースをするとキーは、データはのオブジェクトです

```
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
```

しいをする

shelveをするもなは、**DbfilenameShelf**クラスをするです。 **anydbm**をしてデータをします。クラスをすることも、に**shelve.open**をびすこともできます。

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
finally:
    s.close()
```

データにびアクセスするには、シェルフをいてののようにします。

```
import shelve

s = shelve.open('test_shelf.db')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

のサンプルスクリプトをすると、のようにされます。

```
$ python shelve_create.py
$ python shelve_existing.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

dbmモジュールは、にじデータベースにきむのアプリケーションをサポートしていません。クライアントがシェルフをしないことがわかっているは、**shelve**にデータベースをみりでくようできます。

```
import shelve

s = shelve.open('test_shelf.db', flag='r')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

プログラムがみりでかかれているにデータベースをしようとする、アクセスエラーがされます。タイプは、データベースのにanydbmによってされたデータベースモジュールによってなります。

をく

シェルフは、デフォルトでは、オブジェクトのをしません。つまり、にされているのをしたは、をしてをにするがあります。

```
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()
```

このでは、'key1'にあるはされない、シェルフがかれたときにははされません。

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```


シェルフにされているのオブジェクトへのをにキャッチするには、きしをにしてシェルフをきます。ライトバックフラグにより、シェルフはメモリキャッシュをしてデータベースからりされたすべてのオブジェクトをします。シェルフがじられると、キャッシュオブジェクトもデータベースにきされます。

```
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()
```

プログラマのエラーのをらし、オブジェクトのをよりにすることができますが、ライトバックモードをすることは、あらゆるでましいとはりません。キャッシュは、シェルフがいているになメモリをし、じられたときにキャッシュされたすべてのオブジェクトをデータベースにきすためにすると、ながかかります。キャッシュされたオブジェクトがされたかどうかをするがないので、それらはすべてきされます。アプリケーションがきみよりもくのデータを見る、きしによってにオーバーヘッドがします。

```
$ 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'}
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/10629/>

189:

n を含む iterables のすべてのアルゴリズムは、さが $O(n)$ 。 `bisect.bisect_left()` ようなアルゴリズムだけが、 $O(\log(n))$ さによりになり $O(\log(n))$ 。

Examples

ストリングの **`str.index`**、**`str.rindex`** および **`str.find`**、**`str.rfind`**

String には `index` メソッドだけでなく、よりなオプションとの `str.find` ます。これらのになのがあります。

```
astring = 'Hello on StackOverflow'
astring.index('o') # 4
astring.rindex('o') # 20

astring.find('o') # 4
astring.rfind('o') # 20
```

`index / rindex` と `find / rfind` は、にがつかりません。

```
astring.index('q') # ValueError: substring not found
astring.find('q') # -1
```

これらのメソッドはすべて、インデックスとインデックスをします。

```
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 がつかりません

```
astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - still from left to right

astring.rindex('o', 4, 7) # 6
```

の

Python のみみコレクションはすべて、`in` をつ `in` メンバシップをチェックするをし `in` ます。

リスト

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist # True
10 in alist # False
```

タプル

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple      # False
'4' in atuple    # True
```

```
astring = 'i am a string'
'a' in astring   # True
'am' in astring  # True
'I' in astring   # False
```

セット

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset # True
10 in aset       # False
```

Dict

`dict` はちょっとです。の `in` はキーをチェックするだけです。 `を` するは、 `を` するがあります。 キーとのペアをするもじです。

```
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
```

インデックスリストとタプルの `list.index`、`tuple.index`

`list` と `tuple` は、の `を` する `index` -method があります。

```
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 がリストにありません`

しかし、につかったの `だけ` をします。

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26) # 2
atuple[2]       # 26
atuple[7]       # 26 - is also 26!
```

dictのキーをする

dict はであるため、またはキーをするためのみメソッドがありません。したのキーまたはキーをするをできます。

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[key] == value:
            foundkeys.append(key)
    return foundkeys
```

これは、のリストとしてもくことができます

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

つかったキーが1つだけになる

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

の2つののは、されたをつすべてのkeys list をします。

```
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)  # []
```

もうのキーは1つのキーのみをします。

```
getOneKeyForValue(adict, 10)     # 'c' - depending on the circumstances this could also be 'a'
getOneKeyForValue(adict, 20)     # 'b'
```

StopIteration - がdict ないはException

```
getOneKeyForValue(adict, 25)
```

StopIteration

ソートされたシーケンスのインデックスをするbisect.bisect_left

ソートされたシーケンスでは、よりなアルゴリズムをできます 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)
```

ValueError

にきなされたシーケンスの、のはかなりくなるがあります。に500い1のの

```
%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
```

がのの1つであるはしくなりますが、

```
%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
```

ネストされたシーケンスの

tuple listのようなネストしたシーケンスをするには、dictをキーでするなどののですが、カスタマイズされたがです。

がシーケンスにつかったは、ものシーケンスのインデックス。

```
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
```

またはとののインデックス

```
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
```

に ずしもそうではない `next` と、されたがにするをつジェネレータがもなアプローチです。

カスタムクラスでの `__contains__` および `__iter__`

できるようにするには `in` カスタムクラスのためのクラスはマジックメソッドをしなければならないのいずれか `__contains__` または、そのを、 `__iter__` -method を。

`list` の `list` をむクラスがあるとします。

```
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))
```

`in` をしてメンバーシップテストをすることができます。

```
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__.
```

`__contains__` メソッドをしたでも

```
del ListList.__contains__
5 in a # True
# Prints: Using __iter__.
```

`for i in a` ループ `in` は、クラスが `__contains__` メソッドをしていてもに `__iter__` をします。

オンラインでをむ <https://riptutorial.com/ja/python/topic/350/>

190: の、のいPythonとJavaScript

き

2つのがじまたはしたになるをくことが々こります。のがプログラマにとってい、これらのをにすることは、のをとにするのにちます。

Examples

リストきの`in`

```
2 in [2, 3]
```

Pythonでは、これはTrueにされますが、JavaScriptではfalseにされます。これは、Pythonではがリストにまれているかどうかをチェックするため、2がのとして[2, 3]にあるためです。

JavaScript inはオブジェクトとともにされ、オブジェクトがでされるのプロパティをむかどうかをチェックします。そこで、JavaScriptは、[2,3]をオブジェクトまたはキーマップとみなします。

```
{'0': 2, '1': 3}
```

それにプロパティまたはキー`2`があるかどうかをチェックします。2はかに`2`にされます。

オンラインでの、のいPythonとJavaScriptをむ <https://riptutorial.com/ja/python/topic/10766/の-のい-pythonとjavascript>

191: なヘルプをつCLIサブコマンド

き

`hg`や`svn`のようなサブコマンドをなコマンドラインインターフェイスとヘルプとするは、「」のセクションをしてください。

コマンドラインのでは、のにするよりいトピックがわれます。

ヘルプメッセージにされているコマンドラインインターフェイスをして、`hg`や`svn`のようなサブコマンドをするさまざまな

```
usage: sub <command>

commands:

    status - show status
    list   - print list
```

Examples

ネイティブ・ウェイライブラリなし

```
"""
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())
```

のない

```
usage: sub <command>

commands:
```



```
status - show status
list   - print list
```

- デセッブなし
- もがそれをむことができるはずです
- ヘルプのをに

argparse デフォルトヘルプフォーマッタ

```
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())
```

のない

```
usage: sub {status,list} ...

positional arguments:
  {status,list}
    status      show status
    list        print list
```

- Pythonに
- オプションのがまれています

argparse カスタムヘルプフォーマッタ

されたヘルプの<http://www.riptutorial.com/python/example/25282/argparse--default-help-formatter>のです。

```
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 message
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())

```

のない

```

usage: sub <command>

commands:

```

```
status - show status  
list   - print list
```

オンラインでなヘルプをつCLIサブコマンドをむ <https://riptutorial.com/ja/python/topic/7701>/なヘルプをつcliサブコマンド

192:

き

Pythonは`re`モジュールをしてをにします。

は、ののとしてされるのみわせです。例えば、`'amount\D+\d+'`ワードによってされるのする`amount`などの1つのによってされた `amount=100`、 `amount is 3`、 `amount is equal to: 33`など

-
- `re.match`パターン、 、 フラグ= 0Outのにするパターンまたはなし
- `re.searchpattern`、 `string`、 `flag = 0`Outのパターンとのまたはなし
- `re.findallpattern`、 `string`、 `flag = 0`Outstringまたは[]のpatternのすべてののリスト
- `re.finditerpattern`、 `string`、 `flag = 0`Outre.findallと同じですが、 `iterator`オブジェクトをします。
- パターン、 、 、 フラグ= 0Outパターンのわりにまたはをつ
- プリコンパイルされた
- `precompiled_pattern = re.compile`パターン、 フラグ= 0
- `precompiled_pattern.matchstringOut`のにするかNone
- `precompiled_pattern.searchstringOut`のどこにでもするか、 Noneになります。
- `precompiled_pattern.findallstringOut`するすべてののリスト
- `precompiled_pattern.sub`/パターン/、 Outきえられた

Examples

のにする

`re.match()` ののはで、 2のはするです

```
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'
```

パターンはがのリテラルであることをす`r`であることがわかります。

のリテラルは、リテラルとはしなるをとっています。つまり、のリテラルのバックスラッシュ`\`は"バックスラッシュ"をし、`\n`などのエスケープシーケンスをエスケープするバックスラッシュをするはありません。、タブ`\t`、バックスペース`\`、フォームフィード`\r`などがあります。のリテラルでは、バックスラッシュは、エスケープシーケンスのとみなされることをけるために、2にするがあります。

したがって、`r"\n"`は`\`と`n` 2のす。パターンもバックスラッシュをします。たとえば、`\d`はのをします。の`r"\d"`をして`"\d"`をにエスケープするはあり`"\\d"`。

えば

```
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'
```

マッチングはのからのみされます。どこでもさせたいは、わりに`re.search`をいます

```
match = re.match(r"(123)", "a123zzb")

match is None
# Out: True

match = re.search(r"(123)", "a123zzb")

match.group()
# Out: '123'
```

```
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.'
```

は、`re.match`とはなり、のどこでもされます。 `re.findall`することすもでき`re.findall`。

の`^`ですることすもできます。

```
match = re.search(r"^123", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^123", "a123zzb")
match is None
# Out: True
```

のに \$、

```
match = re.search(r"123$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"123$", "123zzb")
match is None
# Out: True
```

またはをします ^と\$をします。

```
match = re.search(r"^123$", "123")
match.group(0)
# Out: '123'
```

グループピング

グループはカッコでいます。 `group()` びすと、カッコでまれたサブグループにするがされます。

```
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'
```

`group()` にをえて、のサブグループをフェッチすることもできます。

ドキュメントから

が1つの、のはのになります。のがある、そのはごとに1つのをむタプルです。

、 `groups()` をびすと、サブグループをむタプルのリストがされます。

```
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')
```

きグループ

```
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'
```

だけでなくインデックスによってもできるキャプチャグループをします。

グループ

(?:) をするとグループがされますが、グループはされません。つまり、それをグループとしてすることはできますが、あなたの "グループスペース" はされません。

```
re.match(r'(\d+) (\+(\d+))?', '11+22').groups()
# Out: ('11', '+22', '22')

re.match(r'(\d+) (?:\+(\d+))?', '11+22').groups()
# Out: ('11', '22')
```

このではして `11+22` や `11` ではなく `11+`。これは、`+` と `2` がグループされているためです。、`+` はりまれません。

をエスケープする

クラスの `[と]` ようなものはりしません

```
match = re.search(r'[b]', 'a[b]c')
match.group()
# Out: 'b'
```

をエスケープすることで、それらをりさせることができます

```
match = re.search(r'\[b\]', 'a[b]c')
match.group()
```

```
# Out: '[b]'
```

`re.escape()` をってこれをうことができます

```
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'
```

`re.escape()` はすべてのをエスケープするので、ユーザーについてをします。

```
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.!']
```

`re.sub` をしてに `re.sub` ことができます。

の

```
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'
```

グループの

のグループのえは、のようにしてうことができます。

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

ただし、'10'のようなグループIDをした、これはしません。 `\10` は 'ID1のろに0' とみられます。したがって、よりで `\g<i>` をするがあります。

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

の

```
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'
```


すべてしないをする

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Out: ['12', '945', '444', '558', '889']
```

`r"[0-9]{2,3}"`の`r`は、Pythonがをそのままするようにすることにしてください。「の」としていません。

また、することができます `re.finditer()` じようになっている `re.findall()` が、とイテレータをす `SRE_Match` わりにのリストのオブジェクト

```
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
'''
```

プリコンパイルされたパターン

```
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
```

パターンをコンパイルすると、でプログラムでできます。しかし、Pythonはされた [docs](#)、[SOのえ](#) をキャッシュしているので、"をコンパイルすることについてにいくつかのしかないプログラムはするはありません"。

```
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
```

これは`re.match`でできます。

されているのチェック

にのセットこのは`az`、`AZ`および`0-9`のみがまれていることをしたいは、このようにすることができます。

```
import re

def is_allowed(string):
    characterRegex = re.compile(r'^a-zA-Z0-9.>')
    string = characterRegex.search(string)
    return not bool(string)

print (is_allowed("abyzABYZ0099"))
# Out: 'True'

print (is_allowed("#*@#$$%^"))
# Out: 'False'
```

のを`[^a-zA-Z0-9.]` から`[^a-z0-9.]` にして、をすることもできます。

なクレジット <http://stackoverflow.com/a/1325265/2697955>

をしてをする

をしてをすることもできます。えは、

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print( data )
# Output: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

フラグ

なには、のをするがあります。これはフラグをしてわれま。フラグは、`flags` キーワードまたはので2りのでできます。

Flags キーワード

は`re.search`の`re.search`が、`re` モジュールのほとんどのでしま。

```
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'
```

のフラグ

	な
re.IGNORECASE、re.I	のをする
re.DOTALL、re.S	る.をむすべてにマッチする
re.MULTILINE、re.M	MAKES ^ラインのまりと\$のわり
re.DEBUG	デバッグをにする

なすべてのフラグのなリストについては、[ドキュメント](#)をチェックしてください

インラインフラグ

ドキュメントから

(`?iLmsux`) セット 'i'、'L'、'm'、's'、'u'、'x'からの1つの

グループはのとします。re.I、re.Lロケール、re.M、re.Sドット、re.UUnicode、およびするフラグをします。のre.X。これは、フラグをre.compileにすのではなく、のとしてフラグをインクルードするにです。

xフラグは、がどのようにされるかをすることにしてください。これは、にですか、1つののにするがあります。フラグののにがある、はです。

`re.finditer`を使ってマッチをする

re.finditerをすると、のすべてののをできます。これにより、インデックスのするにするなど、のをre.findallするとして、のようになります。

```
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))
```

```
Match "an" found at: [5, 7]
Match "an" found at: [20, 22]
Match "ant" found at: [23, 26]
```

のでのみをさせる

のでのみをマッチさせたいがあります。のをえてみましょう。

```
An apple a day keeps the doctor away (I eat an apple everyday).
```

ここでは、"リンゴ"が2します。これは、しい [regex](#) モジュールでサポートされているいわゆるバックトラッキングでできます。アイデアは

```
forget_this | or this | and this as well | (but keep this)
```

Appleのでは、これはのようになります。

```
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
```

これはのにあるにのみ "apple" にマッチします。

それはどのようにするのです

- からると、エンジンはすべてをにし、`(*SKIP)` は「にのアサーション」としてします。その、しくし`(*FAIL)`、バックトラックします。
- すぐからへ`(*SKIP)`ポイントにります、バックトラック。にくことはじられています。わりに、エンジンはにもてて、`(*SKIP)` がびされたポイントにジャンプするようにされます。

オンラインでをむ <https://riptutorial.com/ja/python/topic/632/-->

193:

- `=` - しくない
- `==` - しい
- `>` - よりきい
- `<` - よりさい
- `>=` - よりきいかしい
- `<=` -
- `is` - オブジェクトがまったくじオブジェクトであるかどうかをテストする
- オブジェクトがまったくじオブジェクトでないはテストしない

パラメーター

パラメータ	
バツ	するの
y	する2の

Examples

よりきいまたはよりさい

```
x > y
x < y
```

これらのは、2つのタイプのをします。の、これはにをしてどのがきいかをします。

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

の、とていますが、アルファベットとていますが、くじではありません。

```
"alpha" < "beta"
# True
```

```
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

これらのは、はよりきいとみなされます。そのため、`"gamma" < "OMEGA"`はです。それらがすべてであった、されるアルファベットののがされます。

```
"GAMMA" < "OMEGA"
# True
```

それぞれの、`<`と`>`をなるですることをしてしているため、するにがのをするものをべるがあります。

しくない

```
x != y
```

`x`と`y`がしくないは`True`し、そうでないは`False`します。

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

にしい

```
x == y
```

このは、`x`と`y`がじであるかどうかをし、をブールとしてします。に、とのがするがあります。したがって、`int 12`は`'12'`とじではありません。

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
# True
'spam' == 'spam '
# False
'12' == 12
# False
```

タイプは、2つのがじかどうかをするためにされるをしなけなければならないことにしてください。みみの、これらのはどおりにし、じについてをするだけです。しかし、カスタムでは、いつでも`True`をすか、に`False`すなど、テストをきなものとしてできます。

チェーンの

チェーンをして、のでのアイテムをできます。え

```
x > y > z
```

のい

```
x > y and y > z
```

のが`True`にのみ`True`とされ`True`。

なはのとおります。

```
a OP b OP c OP d ...
```

`OP`は、できるのの1つをし、はのなをします。

`0 != 1 != 0`は、`0 != 0`が`False`であっても、`True`とされることにしてください。`x != y != z`のなとはなり、`x`、`y`、`z`はなるをちます。はにであるため、`==`はなをもっています。

スタイル

ながあり、するアイテムとのにはのはありません。

```
1 > -1 < 2 > 0.5 < 100 != 24
```

が`True`す、は`True`し`True`。しかし、なをするのはいスタイルではありません。いは`""`であり、よりではありません。

```
1 > x > -4 > y != 8
```

すぐに1つのをすよう`False`、はすぐにされる`False`りのすべてのをスキップし、。

`a > exp > b`の`exp`は1だけされるが、

```
a > exp and exp > b
```

`exp`あればされます `a > exp`です。

``is`vs`==`との`

よくあるとしては、をされ`is`と`==`。

`a == b`のをし、 `a b`。

`a is b`で `a`、 `b`と`b` をする。

する

```
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 [!!]
```

に`is`、 `id(a) == id(b)`とえること`is`できます。

これをえると、ランタイムのがさらにになります。いとさいは、Pythonマシンがじオブジェクトにしてよりないメモリをしようとしているため、 `is`とすると`True`をします。

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

しかし、いときは々にされます。

```
a = 'not so short'
b = 'not so short'
c = 1000
d = 1000
a is b # False
c is d # False
```

あなたはすべき`is`ためにテストすること`None`

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

`is` 「センチネル」すなわちのオブジェクトをテストすることです。


```

sentinel = object()
def myfunc(var=sentinel):
    if var is sentinel:
        # value wasn't provided
        pass
    else:
        # value was provided
        pass

```

オブジェクトの

カスタムクラスのをするために、`__eq__`と`__ne__`メソッドをすることによって、`==`と`!=`オーバーライドできます。あなたはまた、きすることができ`__lt__` <、`__le__` <= `__gt__` >、および`__ge__` >。2つのメソッドをオーバーライドするだけでむことにしてください`==`はりのができます `==`は `not` <や`not` >などとしてです

```

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

```

このでは、`other` のオブジェクトがじオブジェクトであることがです。のタイプとするとエラーがします

```

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'

```

`isinstance()` やそれにするものをチェックすることで、これをぐことができますな。

Common GotchaPythonはタイピングをしません

のくのでは、のようにするとJavaの

```

if("asgdsrf" == 0) {
    //do stuff
}

```

...あなたはエラーがします。そのようなをにとすることはできません。 Pythonでは、これはにです-それだけにます `False`。

よくあるとしはのとおりです

```
myVariable = "1"
if 1 == myVariable:
    #do stuff
```

このは、エラーなしに `False` され、にバグをすか、をるがあります。

オンラインでをむ <https://riptutorial.com/ja/python/topic/248/>

194: らす

- `reduce`、`iterable` [、`initializer`]

パラメーター

パラメータ	
	<code>iterable</code> をらすためにされる2つのをるがあります。のみ
りしな	<code>iterable</code> はするだろう。のみ
イニシャライザ	の。オプション、のみ

`reduce`はにもなであるとはりません。いくつかのには、のまたはメソッドがあります。

- `addable`ではありませんをむシーケンスのの`sum()`

```
sum([1,2,3]) # = 6
```

- ののための`str.join`

```
' '.join(['Hello', ',', ' World']) # = 'Hello, World'
```

- `next` ジェネレータとするとすることができる`reduce`。

```
# First falsy item:
next((i for i in [100, [], 20, 0] if not i)) # = []
```

Examples

```
# 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`ののにりしをすることによって、を`iterable`これまでの。

```
def add(s1, s2):
    return s1 + s2
```

```
asequence = [1, 2, 3]

reduce(add, asequence) # equivalent to: add(add(1,2),3)
# Out: 6
```

このでは、の`add`をしました。しかし、Pythonには、`operator`モジュールにでのがされています。

```
import operator
reduce(operator.add, asequence)
# Out: 6
```

`reduce`はをすることもできます

```
reduce(add, asequence, 10)
# Out: 16
```

reduceをう

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                           arg2=s2,
                                           res=s1*s2))

    return s1 * s2

asequence = [1, 2, 3]
```

`initializer`がえられると、はイニシャライザとの`iterable`にされます。

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30
```

なければ`initializer`パラメータインクルードを`reduce`の2つのリストののをすることによって、を

```
cumprod = reduce(multiply, asequence)
# Out: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
# Out: 6
```

```
import operator
reduce(operator.mul, [10, 5, -3])
# Out: -150
```

すべてのバリエーション

`reduce`は`iterable`がにされるに`iterable`しないので、し`any()`または`all()`をするためにできます。

```
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
```

シーケンスのの1のまたは、もししなければの

```
# 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
```

lambda をするわりに、きをすることがにされています。

```
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])   # = []
```

オンラインでらすをむ <https://riptutorial.com/ja/python/topic/328>/らす

195: モジュール

Examples

のとしての

すべてののは、例えば、そこにある `operator` -function `operator.add` ため、

```
1 + 1
# Output: 2
from operator import add
add(1, 1)
# Output: 2
```

たとえでのみかであるとなでべていますが、

```
from operator import mul
mul('a', 10)
# Output: 'aaaaaaaaaa'
mul([3], 3)
# Output: [3, 3, 3]
```

のPythonドキュメントでは、からへのマッピングもしてください。

メソッドびし

メソッドをにびすこの `lambda` のわりに

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist))      # Keep only elements that start with 'd'
# Output: ['duck']
```

じことをするをうことができます

```
from operator import methodcaller
list(filter(methodcaller('startswith', 'd'), alist)) # Does the same but is faster.
# Output: ['duck']
```

Itemgetter

`itemgetter` をしてのキーとのペアをでグループする

```
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}}
```

これはのような `lambda` とですがよりい

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

または、2のによってタブルのリストをソートするには、のをセカンダリとしてソートします。

```
alist_of_tuples = [(5,2), (1,3), (2,2)]
sorted(alist_of_tuples, key=itemgetter(1,0))
# Output: [(2, 2), (5, 2), (1, 3)]
```

オンラインでモジュールをむ <https://riptutorial.com/ja/python/topic/257/モジュール>

196:

き

ジェネレータは、ジェネレータ `yield` をまたはジェネレータ `(an_expression for x in an_iterator)` をによってされるイテレータです。

- `yield <expr>`
- `<expr>`からのまり
- `<var> = yield <expr>`
- `next <iter>`

Examples

ジェネレータオブジェクトはイテレータプロトコルをサポートします。つまり、Python 3.xの `next()` メソッド `__next__()` をします。これはをステップするためにされ、`__iter__` メソッドはそれをします。これはジェネレータがなiterableオブジェクトをサポートするのでできることをします。

```
# 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]
```

next

`next()` ビルトインは、のイテレータジェネレータイテレータをむからをけり、イテレータがいたされたのデフォルトをするためにできるなラッパーです。

```
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
# ...
```

は `next(iterator[, default])` です。イテレータがし、デフォルトがされたは、それがされます。デフォルトがされていないは、`StopIteration`が`StopIteration`ます。

オブジェクトをジェネレータにする

ジェネレータからをけることにえて、`send()` メソッドをしてジェネレータにオブジェクトをすることもできます。

```
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
```

ここでは、のことがこります。

- に `next(generator)` びすと、プログラムはの `yield` ステートメントにみ、そので `total` の0をします。このでジェネレーターのがされます。
- `generator.send(x)` びすと、インタプリタは `x` をとり、の `yield` のりを `value` にし `value` 。その、ジェネレータは、のがられるまでりをけます。
- に `next(generator)` びすと、プログラムはこれをジェネレーターに `None` をしているかのようにいます。 `None` についてなことはありませんが、このでは、をしようになとして `None` をしています。

ジェネレータの

のようなをしてジェネレータイテレータをすることはです。

```
generator = (i * 2 for i in range(3))
```

```
next(generator) # 0
next(generator) # 2
next(generator) # 4
next(generator) # raises StopIteration
```

がずしもリストをすがないは、ジェネレータをびしのにくことで、をできみやすさをさせることができます。びしのかっこは、をにジェネレータにします。

```
sum(i ** 2 for i in range(4)) # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

さらに、のではのリスト `[0, 1, 2, 3]` をロードするのではなく、ジェネレータはPythonがにじてをできるようにするため、メモリをします。

き

ジェネレータのはlist、dictionary、set comprehensionsとていますが、かっこでまれています。カッコは、びしのものとしてされているときにするはありません。

```
expression = (x**2 for x in range(10))
```

このでは、`0x = 0`をむ10ののなをします。

Generatorは、に1つの`yield`があるをいて、のとています。このようなはを`return`ことはできませんただし、ジェネレータをにさせたいのは`return`をできます。

```
def function():
    for x in range(10):
        yield x**2
```

このジェネレータは、のジェネレータとであり、じものをします。

すべてのジェネレータはそれぞれののをちますが、そのもありません。

のをりすは、なしでジェネレータをできます。

```
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'>
```

のわりに

```
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))
```

だがしかし

```
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)
```

ジェネレータをびすと、ジェネレータオブジェクトがされます。ジェネレータオブジェクトはできます。のタイプのイテレータとはなり、ジェネレータオブジェクトはしかトラバースできません。

```
g1 = function()
print(g1) # Out: <generator object function at 0x1012e1888>
```

ジェネレータのボディはすぐにされないことにしてください。ので`function()`をびすと、の`print`もせずにジェネレータオブジェクトをすぐにします。これにより、ジェネレータは、リストをすよりないメモリをすることができ、にいシーケンスをするジェネレータをすることができます。

このため、ジェネレータは、データサイエンスやのデータをむそののコンテキストでよくされます。のは、なシーケンスがされるのをたずに、のコードがジェネレータによってされたをすぐにできることです。

ただし、ジェネレータでされたをするがあり、コストがのは、シーケンスをするよりも、されたを`list`としてするがいがあります。については、の「ジェネレータのリセット」をしてください。

、ジェネレータオブジェクトは、ループで、またはをとするでされます。

```
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]
```

ジェネレータオブジェクトはイテレータなので、`next()`をとってですることができます。これをうと、のびしごとに1つつがされます。

フードので、ジェネレータで`next()`をびすたびに、Pythonはの`yield`にたるまでジェネレータのでをします。こので、`yield`コマンドのをし、それがこったポイントをしています。`next()`もうびす

と、そのからがされ、の`yield`までできます。

Pythonが`yield`もたずにジェネレータのわりにすると、`StopIteration`がしますこれはですが、すべてのイテレータはじようにします。

```
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
```

Python 2のジェネレータオブジェクトには、されたをでりしするためにできる`.next()`メソッドがあることにしてください。Python 3では、このメソッドはすべてのイテレータにして`__next__()`にきえられました。

ジェネレータのリセット

ジェネレータでされたオブジェクトはしかできないことをえておいてください。にスクリプトのオブジェクトをしているは、それみると`None`がされます。

ジェネレータでされたオブジェクトをするがあるは、ジェネレータをしてするか、またはにするとときにジェネレータのをリストにすることができます。のデータをうは、ジェネレータをすることをおめします。すべてのデータのリストをすると、くのディスクがになります。に、にアイテムをするのにが加かるは、されたアイテムをリストにしてできるようにすることができます。

をってフィボナッチをつける

ジェネレータののは、のをすることです。ここでは、[フィボナッチシーケンス](#)のの10のををつけるをします。

```
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

のシーケンス

ジェネレータは、のシーケンスをすためにできます。

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1
```

```
natural_numbers = integers_starting_from(1)
```

のようなのシーケンスは、 `itertools.count` のけをりてすることもできます。のコードはのよう
にすることができます

```
natural_numbers = itertools.count(1)
```

のジェネレータでジェネレータをすると、しいジェネレータをできます。

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

のジェネレータはしないので、ジェネレータをにしようとするにすと、なになることにしてくだ
さい。

```
list(multiples_of_two) # will never terminate, or raise an OS-specific error
```

わりに、 `range` またはpython <3.0のは `xrange` をしてリスト/の `xrange` をします。

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

`itertools.islice()` にスライスするには `itertools.islice()` をします。

```
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]
```

のジェネレータもじ "ルート" からているのすべてのジェネレータとじようにされていることにし
てください。

```
next(natural_numbers) # yields 16
next(multiples_of_two) # yields 34
next(multiples_of_four) # yields 24
```

のシーケンスは `for` ループをってすることもできます。ループがにするように、き `break` をずめて
ください。

```
for idx, number in enumerate(multiples_of_two):
    print(number)
    if idx == 9:
        break # stop after taking the first 10 multiples of two
```

な - フィボナッチ

```
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
```

のなからすべてののをす

Python 3.x 3.3

のなからすべてののをするは、`yield from`します。

```
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]
```

これはジェネレータもにします。

```
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]
```

コルーチン

ジェネレータは、コルーチンをするためにできます。

```
# 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
```

コルーチンは、ステートマシンをするためによくされます。ステートマシンは、ステートマシンがにするためになシングルメソッドプロシージャをするのにちます。それらはのでし、のにされたをします。

によるディレクトリのすべてのファイルをにリストする

まず、ファイルでするライブラリをインポートします。

```
from os import listdir
from os.path import isfile, join, exists
```

ディレクトリからファイルをみむヘルパー

```
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
```

サブディレクトリのみをするもう1つのヘルパー

```
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
```

これらをして、ディレクトリのすべてのファイルとそのすべてのサブディレクトリジェネレータををにします。

```
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
```

このはyield fromしyield fromできます。

```
def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)
```

ジェネレータをしてする

のジェネレータをするには、`zip`を使います。

```
for x, y in zip(a,b):
    print(x,y)
```

```
1 x
2 y
3 z
```

Python 2では`itertools.izip`わりにうべきです。ここでも、すべての`zip`がタプルをすることがわかります。

`zip`は、iterablesの1つがをいるとすぐにをすることにしてください。もしiterableまでりしたいは、`itertools.zip_longest()`して`itertools.zip_longest()`。

リストコードのリファクタリング

リストからめてりしすることで、リストをしてすなコードがあるとします。

```
def create():
    result = []
    # logic here...
    result.append(value) # possibly in several places
    # more logic...
    return result # possibly in several places

values = create()
```

をリストのにきえることがでないは、をジェネレーターにきえてをすることができます。

```
def create_gen():
    # logic...
    yield value
    # more logic
    return # not needed if at the end of the function, of course

values = list(create_gen())
```

ロジックがであるは、`yield from`をして、びしのすべてのを「フラットされた」にめます。

```
def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)
```


`next`はをしなくてもです。ジェネレータを`next`すことは、あるにマッチするののオカレンスをに
するです。プロセスのようなコード

```
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)
```

のようにきえることができます。

```
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.
```

こののために、 `first = next` ようなエイリアス、またはをするためのラッパーをすることがまし
いがあります。

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/292/>

197:

- `value1 ** value2`
- `pow1、 2 [、 3]`
- `value1 .__ pow __value2 [、 value3]`
- `value2 .__ rpow __value1`
- `operator.pow1、 2`
- `.__ pow __value1、 value2`
- `math.pow1、 2`
- `math.sqrtvalue1`
- `math.expvalue1`
- `cmath.exp1`
- `math.expm11`

Examples

`math.sqrt`および`cmath.sqrt`

`math` モジュールには、`math.sqrt()` `float` できるのをできる `math.sqrt()` がまれています。はに `float` ます。

```
import math

math.sqrt(9)           # 3.0
math.sqrt(11.11)       # 3.3331666624997918
math.sqrt(Decimal('6.25')) # 2.5
```

が `complex`、`math.sqrt()` は `ValueError` させます。

```
math.sqrt(-10)
```

`ValueError` ドメインエラー

`math.sqrt(x)` は `math.pow(x, 0.5)` または `x ** 0.5` よりもですが、のはじです。`cmath` モジュールは、
をすることができ、そののすべてが `+ bi` のであるというをいて、`math` モジュールとにています。
また、`.sqrt()` うこともできます

```
import cmath

cmath.sqrt(4)  # 2+0j
cmath.sqrt(-4) # 2j
```

`j` とはですか `j` は -1 のにします。すべてののは、`a + bi` またはこのは `a + bj` のにれることができます。
`a` は `2+0j` ようなのです。がないので、`b` は `0` です `b` は `2j` の `2` のようにのをします。これにがないので
、`2j` は `0 + 2j` とくこともできます。

みみをったべき**とpow

は、みみのpow functionまたは**をしてできます。

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

ほとんどのすべてのPython 2.xのでは、のはよりいオペランドのになります。これは**はてはまりません。のケースはこのルールのです。

- ベース int 、 int < 0

```
2 ** -3
# Out: 0.125 (result is a float)
```

- これはPython 3.xでもです。
- Python 2.2.0よりは、 ValueError しました。
- int < 0またはfloat < 0 、 float != int

```
(-2) ** (0.5) # also (-2.) ** (0.5)
# Out: (8.659560562354934e-17+1.4142135623730951j) (result is complex)
```

- Python 3.0.0よりは、 ValueError しました。

operatorモジュールには、 ** operatorとの2つのがまれています。

```
import operator
operator.pow(4, 2)      # 16
operator.__pow__(4, 3)  # 64
```

または__pow__メソッドをびすことができます

```
val1, val2 = 4, 2
val1.__pow__(val2)      # 16
val2.__pow__(val1)      # 16
# in-place power operation isn't supported by immutable classes like int, float, complex:
# val1.__ipow__(val2)
```

mathモジュールをしたmath.pow

math math.pow()のはmath.pow()がまれています。みみのpow() functionまたは**とのいは、かにfloatであることです。

```
import math
math.pow(2, 2)          # 4.0
math.pow(-2., 2)        # 4.0
```

なをうをする

```
math.pow(2, 2+0j)
```

`TypeErrorcomplexをfloatにできません`

なにびつくと

```
math.pow(-2, 0.5)
```

`ValueErrorドメインエラー`

math.expおよびcmath.exp

`math`と`cmath`のモジュールには、[Eulerの](#)がまれています。eをみみ`pow()` functionまたは`**`-operator すると、`math.exp()`ようにします。

```
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)
```

しかし、はなりません。をすると、`math.e`みみよりもにがくなり`math.e`。

```
print(math.e ** 10) # 22026.465794806703
print(math.exp(10)) # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
# difference starts here -----^
```

マイナス1math.expm1

`math`モジュールには、`expm1()` `math.exp(x)` または `cmath.exp(x)` よりものいにさい`x`について`math.e ** x - 1`をできる`expm1()`がまれています。

```
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
# -----^
```

にさい`x`の、はきくなります。

```
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
```

```
# ^-----
```

これは、においてである。たとえば、[プランク](#)のには、から1をいたものがまれます。

```
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.expml(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
#                               ^-----
```

マジックメソッドとみみ、と**cmath**

なをするクラスがあるとします。

```
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)
```

みみの`pow`または`**`をすると、に`__pow__`がびされ`__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__
```

`__pow__()` メソッドの2のは、`__pow__()` `pow()` をうか、メソッドをびすことによってのみえられます

```
pow(Integer(2), 3, 4)           # Integer(0)
# Prints: Using __pow__ with modulo
Integer(2).__pow__(3, 4)        # Integer(0)
# Prints: Using __pow__ with modulo
```

`math` はに `float` し、`float` をします。

```
import math

math.pow(Integer(2), 0.5) # 1.4142135623730951
# Prints: Using __float__
```

`cmath` -functions はそれをしようと `complex` だけでなく、フォールバックすることができます `float` へのながしないは `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__
```

`__float__()` も `__float__()` ない、`math` も `cmath` も `__float__()` ません

```
del Integer.__float__         # Deleting __complex__ method

math.sqrt(Integer(2))         # also cmath.exp(Integer(2))
```

TypeError がです

モジュラ **3** つのを **pow**

`pow(a, b, c)` `pow()` を3つの `pow()` `pow(a, b, c)` べき $a^b \bmod c$

```
pow(3, 4, 17)                # 13

# equivalent unoptimized expression:
3 ** 4 % 17                   # 13

# steps:
3 ** 4                        # 81
81 % 17                       # 13
```

モジュラをするみみのは、のにのみです。

- のは `int`
- 2のは `int >= 0`
- 3のは `int != 0`

これらのは、Python 3.xにもあります

たとえば、3つのをとる `pow` をして、モジュラをすることができます。

```
def modular_inverse(x, p):  
    """Find a such as  $a \cdot x \equiv 1 \pmod{p}$ , 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]
```

き n

`math.sqrt` はののにされますが、のような n をするには、`**` をののであるとなることがよくあります。

のはのによるです。したがって、3のにくことによってをてることができれば、そのを $1/3$ のにくことによって、のをつけることができます。

```
>>> x = 3  
>>> y = x ** 3  
>>> y  
27  
>>> z = y ** (1.0 / 3)  
>>> z  
3.0  
>>> z == x  
True
```

き n のルーツをする

Pythonはネイティブにき n をサポートしていますが、にき n の n のルーツをすることはPythonではするがあります。

```
x = 2 ** 100  
cube = x ** 3  
root = cube ** (1.0 / 3)
```

`OverflowError: long intがきすぎてfloatにできない`

そのようなき n をう、カスタムをしての n のルーツをするがあります。

```
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
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/347/>

198: な

- `cmath.rectAbsoluteValue`、`Phase`

Examples

な

モジュール `cmath` は、をするためのがまれています。

```
import cmath
```

このモジュールはのをラジアンでできます

```
z = 2+3j # A complex number
cmath.phase(z) # 0.982793723247329
```

それはのデカルトととののをにする

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)
cmath.rect(2, cmath.pi/2) # (0+2j)
```

このモジュールには、

- 、いつものように、`log` であり `log10`。

```
cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)
cmath.log(z) # (1.2824746787307684+0.982793723247329j)
cmath.log10(-100) # (2+1.3643763538418412j)
```

- `cmath.sqrt(z)` # (1.6741492280355401+0.8959774761298381j)

- とその

```
cmath.sin(z) # (9.15449914691143-4.168906959966565j)
cmath.cos(z) # (-4.189625690968807-9.109227893755337j)
cmath.tan(z) # (-0.003764025641504249+1.00323862735361j)
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)
```

- とその

```
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
```

Pythonにはながみまれています。を`j`とする。

```
z = 2+3j # A complex number
w = 1-7j # Another complex number
```

は、、、、およびすることができます。

```
z + w # (3-4j)
z - w # (1+10j)
z * w # (23-11j)
z / w # (-0.38+0.34j)
z**3 # (-46+9j)
```

Pythonは、のとをし、とをすることもできます。

```
z.real # 2.0
z.imag # 3.0
abs(z) # 3.605551275463989
z.conjugate() # (2-3j)
```

オンラインでなをむ <https://riptutorial.com/ja/python/topic/1142>/な

199:

- `mydict = {}`
- `mydict [k] =`
- `= mydict [k]`
- `= mydict.getk`
- `value = mydict.getk、 "default_value"`

パラメーター

パラメータ	
キー	するのキー
	またはす

をするときにえておくべきつ

- すべてのキーはでなければなりませんそうでないはオーバーライドされます
- すべてのキーハッシュでなければなりませんができる`hash`それをハッシュするを、そうでない`TypeError`スローされます
- キーのはにありません。

Examples

のへのアクセス

```
dictionary = {"Hello": 1234, "World": 5678}
print(dictionary["Hello"])
```

のコードは1234ます。

このの"Hello"はキーとされます。これは、キーをで`dict`によって、`dict`をするためにされます。

1234は、`dict`のそれぞれのコロンのにされます。これは"Hello"がこの`dict` マップする とされます。

しないキーでこのようなを`KeyError`がし、キャッチされないはをします。`KeyError`をにさらすことなくにアクセスしたいは、`dictionary.get`メソッドをできます。デフォルトでは、キーがしない、メソッドは`None`をします。ルックアップにしたは`None`わりに2のをしてすことができます。

```
w = dictionary.get("whatever")
x = dictionary.get("whatever", "nuh-uh")
```

このでは、`w`は`None`をし、`x`は`"nuh-uh"`をします。

dictコンストラクタ

`dict()` コンストラクタをして、キーワードから、またはキーとのペアを1つずつりして、またはとキーワードからをできます。

```
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}
```

KeyErrorのをける

をするときのなとしの1つは、しないキーにアクセスすることです。これは、`KeyError`をさせます

```
mydict = {}
mydict['not there']
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

キーエラーをする1つののは、`dict.get`メソッドをすることです。このメソッドでは、キーのにされるデフォルトをできます。

```
value = mydict.get(key, default_value)
```

するは`mydict[key]`をしますが、それのは`default_value`します。これは`mydict`に`key`をしないことにしてください。あなたがそのキーとのペアをしたいのであれば、あなたがするがあります

`mydict.setdefault(key, default_value)` キーとのペアをしません、。

```
mydict = {}
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}
print(mydict.setdefault("foo", "bar"))
# bar
print(mydict)
# {'foo': 'bar'}
```

このにするのは、をキャッチすることです

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

また、キーが`in`あるかどうかをすることもできます。

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

ただし、マルチスレッドでは、チェックしたにキーをからして、がスローされるのあるをするがあります。

のオプションは、`dict`のサブクラスである`collections.defaultdict`をし、`new_key`がえられたときに`dict`にしいエントリをする`default_factory`をつことです。

キーとへのアクセス

をうときは、`for`ループ、リストの、またはなリストのいずれかで、のすべてのキーとにアクセスするがあることがよくあります。

のようながあるとします。

```
mydict = {
    'a': '1',
    'b': '2'
}
```

`keys()` メソッドをして `keys()` リストをることができます

```
print(mydict.keys())
# Python2: ['a', 'b']
# Python3: dict_keys(['b', 'a'])
```

わりに`values`のリストがなは、`values()` メソッドをし`values()`。

```
print(mydict.values())
# Python2: ['1', '2']
# Python3: dict_values(['2', '1'])
```

キーとそれにするのをするは、`items()` メソッドをできます。

```
print(mydict.items())
# Python2: [('a', '1'), ('b', '2')]
# Python3: dict_items([('b', '2'), ('a', '1')])
```

`dict`はソートされていないため、`keys()`、`values()`、および`items()`にはソートはありません。これらのメソッドがすがになるは、`sort()`、`sorted()`、`OrderedDict`し`sort()`。

Python 2/3のい Python 3では、これらのメソッドはリストではなくなりしなオブジェクトをします。これはPython 2 `iterkeys()`、`itervalues()`、`iteritems()`メソッドと`iteritems()`です。これらのオブジェクトは、ほとんどの、リストのようにできますが、いくつかのいがあります。は、

PEP 3106をしてください。

の

ディクショナリは、Pythonでのマッピングともばれるキーストアのです。これにより、キーをしてをおよびすることができます。はキーでされるため、はにです。にキーによるのにされるため、ソートされません。

dictをする

はさまざまなですることができます。

リテラル

```
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}
```

の

```
d = {k:v for k,v in [('key', 'value',)]}
```

みみクラス dict()

```
d = dict() # empty 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)
```

dictをする

アイテムをにするには、をしてしいキーをするだけです。

```
d['newkey'] = 42
```

として `list` と `dictionary` をすることもできます

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

アイテムをするには、からキーをします。

```
del d['newkey']
```

デフォルトをつ

`defaultdict` としてライブラリで

```
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'
```

[*] `dict.setdefault()` をすると、みみの `dict` クラスをするがある、にしまったキーにアクセスするたびにデフォルトをすることができます。

```
>>> d = {}
{}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
{'Another_key': ['This worked!']}
```

するがたくさんある、`dict.setdefault()` はびされるたびにこのでは `[]` のしいインスタンスをし、なワークロードをするがあることに `dict.setdefault()` 。

[*] *David Beazley* と *Brian K. Jones* O'Reilly による *Python Cookbook*、3。 Copyright 2013 David Beazley and Brian Jones、978-1-449-34037-7。

きの

ディクショナリのキーをするときに、されたにうきをできます。

`collections` モジュールから `OrderedDict` をします。これは、りしされたときのオーダーのをにします。

```
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])
```

**をしてをする

**キーワードアンパックをすると、のキーとのペアをのにすことができます。 [のな](#)

```
>>>
>>> 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 !
```

Python 3.5では、このをしてのdictオブジェクトをマージすることもできます。

```
>>> 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'}
```

このですように、キーはのにマッピングされますたとえば、「Clifford」は「Nemo」をきします。

をマージする

のをえてみましょう。

```
>>> 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'}
```

このですように、キーはのにマッピングされますたとえば、「Clifford」は「Nemo」をきします

Python 3.3

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

このでは、の "Clifford"が "Nemo"のためにてられますではなく、されたキーにするがされます。

Python 2.x、3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

これは、マージのための**ベースのテクニック "Clifford"が "Nemo"をきするのように、のをします。

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update`はのdictをしてのdictをきします。

のカンマ

リストやタプルのように、にカンマをにめることができます。

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

PEP 8では、のカンマとじたかっこのにスペースをれておくがあります。

のすべてのみわせ

```
options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}
```

のようながあれば、するキーをべるためにのをすリストがあります。 "x"="a" "y"=10 、に "x"="a" "y"=10 、すべてのなみわせをするまでしたいとします。

のコードをして、そのようなのみわせをすべてすリストをできます。

```
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
```

これにより、の combinations されているのリストがられます。

```
[{'x': 'a', 'y': 10},
 {'x': 'b', 'y': 10},
 {'x': 'a', 'y': 20},
 {'x': 'b', 'y': 20},
 {'x': 'a', 'y': 30},
 {'x': 'b', 'y': 30}]
```

のりし

をイテレータとしてする for ステートメントなど、のキーをします。えば

```
d = {'a': 1, 'b': 2, 'c': 3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
```

してもじことがえます

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x 3.0

items() メソッドをすると、キーとのをにループすることができます。

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

values() メソッドをして、だけをするすることができます。

```
for key, value in d.values():
    print(key, value)
```

```
# 3
# 2
# 1
```

Python 2.x 2.2

ここでは `keys()`、`values()`、および `items()` メソッドがリストをします `iterkeys()` `itervalues()`、`iteritems()` の3つのメソッドがあります。

の

をするためのルール

- すべてのキーはでなければなりませんそうでないはオーバーライドされます
- すべてのキーハッシュでなければなりませんができる `hash` それをハッシュするを、そうでない `TypeError` スローされます
- キーのはにありません。

```
# 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}
```

の

はキーをにマップします。

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

ディクショナリは、キーでアクセスできます。

```
print "Little " + car["color"] + " " + car["model"] + "!"  
# This would print out "Little Red Corvette!"
```

は、JSONですることもできます。

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

ディクショナリはのようになりしできます。

```
for key in car:  
    print key + ": " + car[key]  
  
# wheels: 4  
# color: Red  
# model: Corvette
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/396/>

200: の

き

ここでスタックオーバーフローでは、じエラーについてしていることがよく "ImportError: No module named '?????'"。 "ImportError: No module named '?????'"、 NameError: name '???' is not defined SyntaxError: invalid syntax NameError: name '???' is not defined または NameError: name '???' is not defined。 これはそれらをらし、リンクするためのドキュメントをするためのです。

Examples

IndentationErrors または インデントの SyntaxErrors

のほとんどのでは、インデントはではありませんが、PythonおよびのFORTRAN、Makefiles、Whitespaceなどではそうではありません。あなたがのからた、コードをサンプルからのコードにコピーしていた、またはにしいコードであれば。

IndentationError / SyntaxError しない インデント

このは、インデントレベルがし、がないにします。

ここでレベルをげるはありません。

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")
```

ここには2つのエラーがあります。のものとインデントがインデントレベルにしないことです。ただ1つしかされません

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 インデントがのインデントレベルとしません

あなたはにしていないようにえます。

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 インデントされたブロックがです

コロンそしてしいの、インデントレベルがしなければなりません。このエラーはしなかったにします。

```
if ok:
doStuff()
```

キーワードをし`pass`にくためにそれはにもいません `if`、`else`、`except`、`class`、`method` または `definition` なく、ばれる/がしているとき、がこるかとうただし、でそれをう、またはの、`except` もしない

```
def foo():
    pass
```

IndentationError インデントでのタブとスペースのののない

```
def foo():
    if ok:
        return "Two != Four != Tab"
        return "i dont care i do whatever i want"
```

このエラーをする

タブをしないでください。PythonのスタイルガイドであるPEP8ではおめできません。

1. インデントに4つのスペースをするようにエディタをします。
2. とをい、すべてのタブを4つのスペースにきえます。
3. エディターがタブを8つのスペースとしてするようにされていることをしてください。そうすれば、にエラーをすることができます。

については、[この](#)をしてください。

TypeError

これらは、のオブジェクトのタイプになるがあるにします

TypeError[/メソッド]はかかりますかなだけとえられた

またはメソッドがけれなよりもくまたはないでびされました。

よりくのがえられた

```
def foo(a): return a
foo(a,b,c,d) #And a,b,c,d are defined
```

よりないがえられた

```
def foo(a,b,c,d): return a += b + c + d
foo(a) #And a is defined
```

のをするは、`*args`または`**kwargs`できます。[* args](#)と[** kwargs](#)をしてください

TypeError[operand]のサポートされていないオペランドタイプ '???'と '???'

オペランドによっては、のタイプをにすることはできません。

たとえば、`+`はおよびにされますが、どちらのタイプにもすることはできません。たとえば、`'set1'`と`'tuple1'`を`+`ingして`set`をしようとする、エラーがします。コード

```
set1, tuple1 = {1,2}, (3,4)
a = set1 + tuple1
```

いくつかの `int` や `string` は`+`うが、

```
b = 400 + 'foo'
```

あるいは、にもわれないかもしれません。

```
c = ["a", "b"] - [1,2]
```

しかし、えば、`float`を`int`することができます

```
d = 1 + 1.0
```

TypeError '???'オブジェクトはiterable / subscriptableではありません

オブジェクトをにするには、ゼロからしてインデックスがもはやでなくなり、`IndexError`がするまで、したインデックスをすることができますには`__iter__`をす`__getitem__`メソッドをた`__getitem__`ばなりませ`__iterator__`、または`__getitem__`べたもの。

ここでは、`bar`は1の0のだとしてい`bar`。ナンセンス

```
foo = 1
bar = foo[0]
```

もっとしたバージョンですこの`for`は`x`を`amount[0]`にしようとしています、`iterable`なのですが、`amount`は`int`なのでできません

```
amount = 10
for x in amount: print(x)
```

TypeError '???'オブジェクトはびしではありません

せん

をしてでびすやメソッドでうことのように

```
foo = "notAFunction"
foo()
```

NameError '???'されていません

されていない、メソッド、をしようとしたときにしますなくともはありません。つまり、されたローカルまたはグローバルが見つからないにします。オブジェクトのをえたり、かを `import` するのをれたがあります。それはのにあるかもしれません。のでそれらをカバーします。

それはにコードのどこにもされていません

あなたがするのをれているがあります、にそれがの

```
foo    # This variable is not defined
bar()  # This function is not defined
```

それはでされるでしょう

```
baz()

def baz():
    pass
```

それとも、 `import` されていない

```
#needs import math

def sqrt():
    x = float(input("Value: "))
    return math.sqrt(x)
```

Python スコープとLEGBルール

いわゆるLEGBルールは、Pythonのスコープについてります。それはなるスコープについており、のによってけられています。

```
Local → Enclosed → Global → Built-in.
```

- **L**ocal グローバルにされていないか、にされていない。
- **E**nclosing ののにラップされたでされた。
- **G**lobal グローバルにされた、またはファイルのにりてられた。
- **B**uilt-in みみのモジュールにあらかじめりてられた。

として

```
for i in range(4):
    d = i * 2
print(d)
```

d はアクセスです。これは、`for` ループがしいスコープをマークしないためですが、そうした、エラーがし、そのはのようになります。

```
def noaccess():
    for i in range(4):
        d = i * 2
noaccess()
print(d)
```

Python は `NameError: name 'd' is not defined` とって `NameError: name 'd' is not defined`

そののエラー

AssertError

`assert` ステートメントはほとんどすべてのプログラミングでします。あなたがするとき

```
assert condition
```

または

```
assert condition, message
```

これはこれとじです

```
if __debug__:
    if not condition: raise AssertionError(message)
```

アサーションにはオプションのメッセージをめることができ、デバッグがしたらアサーションをにすることができます。

みみのデバッグは、のでは `True`、がされると `False` コマンドラインオプション `-O` です。デバッグのりてはです。インタープリターがすると、みみのがされます。

KeyboardInterrupt

ユーザーがリミキー、は`Ctrl + C`または`del`キーをしたときにするエラー。

ZeroDivisionError

されていない`1/0`をしようとしてしました。のをつけるには、のをしてください。

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)
```

`for`ループがそのを`x`りてるため、`ZeroDivisionError`が`for`ます。わりにそれがです

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)
```

コードのエラー

のないをしす`SyntaxError`は、ののにそののにがあることをしますこのではありません。

```
def my_print():
    x = (1 + 1
    print(x)
```

```
File "<input>", line 3
    print(x)
    ^
```

```
SyntaxError: invalid syntax
```

こののものは、にされているように、カッコ/カッコのです。

Python 3ではprintステートメントのなが1つあります。

Python 3.x 3.0

```
>>> print "hello world"
File "<stdin>", line 1
    print "hello world"
          ^
SyntaxError: invalid syntax
```

printステートメントはprint()にきえられているので、のようにします。

```
print("hello world") # Note this is valid for both Py2 & Py3
```

オンラインでのをむ <https://riptutorial.com/ja/python/topic/9300/>の

201:

き

あなたがOOPからたかどうにかかっているように、クラスをしてすることは、コードをくときにすべきです。

なるバージョンをするためにをし、それをするのができたらどうでしょうかそれをのシナリオでできるようにのパラメータをバインドする、あるのとえます。

- 、** params_you_want_fix

パラメーター

Param	
バツ	てる
y	
げる	する

Pythonのdocにあるように*functools.partial*

びされたときに、argsとキーワードargumentsキーワードでびされるfuncのようにする
しいオブジェクトをします。びしにもっとくのがえられた、それらはargsにされます
。のキーワードがされた、それらはキーワードをしてきします。

なをするには、 [このリンク](#) をチェックしてください。

Examples

をげる

xをyにすることをえてみましょう。

あなたはこれをのようになっています。

```
def raise_power(x, y):  
    return x**y
```

あなたのyのがののをることができるなら、どうでしょうか

y が $[3,4,5]$ の1つであるとしましょう。エンドユーザーににがいので、そのようなをするをんでいないとしましょう。には、された y がなになっているかどうかをチェックし、をのよういきします。

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

メッシーなフォームをして、3つすべてのケースにしましようにしましう。

```
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)
```

そこでがこるの y パラメータをし、3つのなるをしました。

でしたをするはありませんがプライベートにすることもできます、にされたをしてをにすることができます。

オンラインでをむ <https://riptutorial.com/ja/python/topic/9383/>

き

Pythonの「」は、CやJavaのようなプログラミングではなく、リストにいます。リストは、またはのコレクションにすることができ、int、またはのリストをむことができます。

パラメーター

パラメータ	
b	サイズ1バイトのきをします。
B	サイズ1バイトのなしをす
c	サイズ1バイトのをします。
u	サイズ2バイトのUnicodeをします。
h	サイズが2バイトのきをします。
H	2バイトのなしをす
i	サイズが2バイトのきをします。
I	2バイトのなしをす
w	サイズが4バイトのUnicodeをします。
l	サイズが4バイトのきをします。
L	サイズが4バイトのなしをす
f	サイズが4バイトのをします。
d	サイズが8バイトのをします。

Examples

のな

は、じデータののをするデータです。Pythonでは、これがとリストのないます。

Pythonリストには、なるデータにするをめることができますが、Pythonのにはじデータにするし

かめることができません。このチュートリアルでは、いくつかのがあるPythonをします。

Pythonをめておいのは、Pythonのはじめのからめましょう。

Pythonでをするには、の`array`モジュールをインポートするがあります。これは、が、などのなデータではないためです。ここでは、Pythonで`array`モジュールをインポートするをします

```
from array import *
```

`array`モジュールをインポートしたら、`array`をできます。あなたのやりはのとおりです

```
arrayIdentifierName = array(typecode, [Initializers])
```

のでは、`arrayIdentifierName`はの、`typecode`はのをPythonにらせ、`Initializers`はが`Initializers`されるです。

コードは、のまたはのをするためにされるコードです。パラメータセクションのテーブルは、とそのをするときにできるをしています。

Pythonのののをにします

```
my_array = array('i', [1,2,3,4])
```

のでは、されるタイプコードは`i`です。このコードは、サイズが2バイトのきをします。

ここでは、5つのをむのなをします

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

インデックスをして々のにアクセスする

々ののは、をしてアクセスできます。Pythonはゼロインデックスされています。にをします。

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
print(my_array[2])
# 3
print(my_array[0])
# 1
```


append メソッドをしてにをする

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

6がのにされていることにしてください。

insert メソッドをしてにをする

insert() メソッドをして、ののインデックスにをできます。にをします。

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

のでは、0がインデックス0にされています。のはインデックスで、2のはです。

extend メソッドをってPythonをする

Pythonは、 extend() メソッドをしてのでできます。にをします。

```
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])
```

my_arrayはmy_extnd_arrayでされています。

fromlist メソッドをしてリストからをにする

にをします。

```
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])
```

そこで、11,12、および13がリストcからmy_arrayされたことがmy_arrayます。

remove メソッドをしてをする

にをします。

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

4がからされたことがわかります。

pop メソッドをしてのをする

pop はからのをします。にをします。

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

したがって、の 5 がからポップされたことがわかります。

index メソッドをして、そのインデックスをじてのをします。

index() は、するののインデックスをします。はゼロインデックスけされていることにしてください。

```
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
```

2のでは、がに2しても、インデックスが1つだけされたことにしてください

reverse メソッドをしてPythonをにする

reverse() メソッドは、がをするかをします。つまり、をにします。にをします。

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

buffer_info メソッドをしてバッファをする

このメソッドは、メモリのバッファのアドレスとarrayののをします。にをします。

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

count メソッドをしてのをチェックする

count() はcount() をし、はにれます。のでは、3が2することがわかります。

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
```

tostring メソッドをしてをにする

tostring() はをにします。

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

tolist メソッドをしてじをつをpython リストにする

Python list オブジェクトがなときは、tolist() メソッドをつてをリストにすることができます。

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

fromstring メソッドをしてcharにをする

fromstring() をしてにをできます。

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

オンラインでをむ <https://riptutorial.com/ja/python/topic/4866/>

203:

Examples

Condaパッケージのアップロード

めるに、のものがです。

システムにインストールされているAnaconda Binstarのアカウント [Anaconda 1.6](#)をしていないは、 [binstar](#) コマンドラインクライアントをインストールします。

```
$ conda install binstar
$ conda update binstar
```

Anacondaをしていない、Binstarはpypiでもできます。

```
$ pip install binstar
```

これでログインできます

```
$ binstar login
```

whoamiコマンドでログインをテストしてください

```
$ binstar whoami
```

私たちはシンプルな「こんにちは」のをってパッケージをアップロードします。 Githubからのデモンストレーションパッケージのレポをすることからめるには

```
$ git clone https://github.com/<NAME>/<Package>
```

これはのようなさなディレクトリです。

```
package/
  setup.py
  test_package/
    __init__.py
    hello.py
    bld.bat
    build.sh
    meta.yaml
```

Setup.pyはのpythonビルドファイルで、 hello.pyはのhello_worldをとっています。

bld.bat、 build.sh、 およびmeta.yamlのためのスクリプトとメタデータです Condaパッケージ。 これらの3つのファイルとそののについては、 [Condaビルド](#) ページをむことができます。

に、のコマンドをしてパッケージをします。

```
$ conda build test_package/
```

コンダパッケージのにはこれがすべてです。

のステップは、`conda build test_package /`コマンドをした、のをコピーしてりけてbinstarにアップロードすることです。のシステムでは、コマンドは

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

パッケージをしてめてリリースするので、ウェブアプリケーションをしてテキストフィールドをするがあります。

CondaパッケージをBinstarににアップロードしたことをするために、`done`ことがされます。

オンラインでをむ <https://riptutorial.com/ja/python/topic/4064/>

204:

き

Pythonのは、ののアクションをするための、されたなモジューコードをします。はコーディングプロセスをし、ロジックをし、コードをにフォローします。このトピックでは、Pythonでののとについてします。

Pythonには、`print()`、`input()`、`len()`などのみみがあります。みみのに、のをしてよりのジョブをすることもできます。これはユーザーとばれます。

- `def function_name arg1、... argN、* args、kw1、kw2 =デフォルト、...、** kwargs` ステートメント
- ラムダ`arg1、... argN、* args、kw1、kw2 =デフォルト、...、** kwargs`

パラメーター

パラメータ	
<code>arg1、...、argN</code>	の
<code>* args</code>	の
<code>kw1、...、kwN</code>	キーワードのみの
<code>** kwargs</code>	りのキーワード

でできる5つのなこと

- にをりてる

```
def f():  
    print(20)  
y = f  
y()  
# Output: 20
```

- ののをする **れ**

```
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
```

- はのをすことができる

```
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
```

- はのにパラメータとしてすことができます

```
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
```

- は、みスコープ [Closure](#) にアクセスできます。

```
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!
```

そののリソース

- とデコレータの <https://www.thecodeship.com/patterns/guide-to-python-function-decorators/>

Examples

なのとびし

`def` をするのは、Pythonでをするもなです。このは、のをつ、いわゆるです。

```
def function_name(parameters):
    statement(s)
```

`function_name` は、のとしてられています。はなステートメントなので、をオブジェクトにバインドします。オブジェクトはでそのをしてびすことができます。

`parameters` は、がびされたときにしてされたにバインドされるのオプションのリストです。は、コンマでられたののをつことができます。

`statement(s)` ともばれますは、がびされるたびにされるのでないシーケンスです。つまり、[インデントされたブロック](#)とに、をにすることはできません。

に、びされるたびにHelloをするなのをします。

```
def greet():  
    print("Hello")
```

に、されたgreet()をびしましょう

```
greet()  
# Out: Hello
```

これはのもう1つので、がびされるたびに1つのをとり、されたをします。

```
def greet_two(greeting):  
    print(greeting)
```

その、をしてgreet_two()をびすがあります。

```
greet_two("Howdy")  
# Out: Howdy
```

また、そののにデフォルトをえることもできます

```
def greet_two(greeting="Howdy"):  
    print(greeting)
```

これで、をえずにをびすことができます

```
greet_two()  
# Out: Howdy
```

のくのとはなり、のりのをにするはありません。Pythonは、returnキーワードでののをすことができます。1つのは、ののなるをすことができます

```
def many_types(x):  
    if x < 0:  
        return "Hello!"  
    else:  
        return 0  
  
print(many_types(1))  
print(many_types(-1))  
  
# Output:  
0  
Hello!
```

これがびしによってしくされているり、これはになPythonコードです。

returnなしでのにするは、にNoneします。


```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

のように、には、つまりでないのステートメントがです。したがって、`pass`はとしてされます。これは`null`です。されるところありません。それがすることは、それはスキップします。がになどきにプレースホルダとしてですが、コードをするはありません。

からのり

は、できるを`return`ことができます。

```
def give_me_five():
    return 5

print(give_me_five()) # Print the returned value
# Out: 5
```

またはですのためにをします。

```
num = give_me_five()
print(num) # Print the saved returned value
# Out: 5
```

のにをします。

```
print(give_me_five() + 10)
# Out: 15
```

ファンクションで`return`がした、ファンクションはにし、のオペレーションはされません。

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

タブルのでのを`return`することもできます

```
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
```

`return`

がないは、に `None` します。に、 `return` をち、りももないは、 `None` します。

をつをする

は、ののカッコにされています。

```
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)
```

とそのリストは、のとされます。きはのローカルです。

をびすときは、をにべてします

```
divide(10, 2)
# output: 5
```

のをしてのすることができます。

```
divide(divisor=2, dividend=10)
# output: 5
```

オプションのをつをする

オプションのは、`argument-name`に `=`をしてデフォルトをすることでできます。

```
def make(action='nothing'):
    return action
```

このをびすは3つあります。

```
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
```

な `list`、`dict`、`set`などは、デフォルトとしてえられたときにはがです。デフォルトのをするとにされます。 [オプションのをしたのを](#)してください。

のをつをする

1つののは、なのをえることができます。のされたは、がでなければならず、なは、

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

をびすと、のないキーワードをえることができますが、はです。

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

あるいは、をnameとwithoutをみわせてみわせます。それでのついたものは、それらのものにわなくてはならないが、のついたもののはではない。

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

ののをつをする

のの

ののをることができるをするには、の1つに*けるか、

```
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
```

func(*args=[1, 2, 3])ようにfunc(*args=[1, 2, 3]) argsデフォルトをすることはできません コンパイルされなくても。

をびすときにこれらのをすることはできません。たとえばfunc(*args=[1, 2, 3])はTypeErrorます。

しかし、またはのIterable にをすでになっているは、 func(*my_stuff)ようにをびすことができます。

これらの `*args` は、インデックスによってアクセスできます。例えば、`args[0]` はのをしします

キーワードの

のついたののをすることができます。そののに2つの `*` をけたのをしします。

```
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
```

をせずにこれらをすることはできません。たとえば、`func(1, 2, 3)` は `TypeError` ます。

`kwargs` はなネイティブのpythonです。たとえば、`args['value1']` は `value1` を `value1` ます。そのよう
ながあるか、`KeyError` が `KeyError` することをにしてください。

これらをのオプションとなとさせることはできますが、のはです。

1. キーワードがにます。 の。

そして、の `*arg` がます。 オプション。

に、キーワードのみののがにる。 。

にのキーワード `**kwargs` がる。 オプション。

```
#      |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10, *args, kwarg1, kwarg2=2, **kwargs):
    pass
```

- `arg1` があります。そうでないは、`TypeError` が `TypeError` されます。それは、`func(10)` または
キーワード `func(arg1=10)` としてえることができます。
- `kwarg1` も `kwarg1` がありますが、キーワード `func(kwarg1=10)` としてのみできます。
- `arg2` と `kwarg2` はオプションです。をするは、`arg1` またはキーワードのいずれかおよび `kwarg1`
キーワードのみとじルールがされます。
- `*args` のパラメータをしします。しかし、`*args func(1, 1, 1, 1) *args` をすためには、`arg1` と
`arg2` をとしてするがあります。
- `**kwargs`

すべてののキーワードパラメータをキャッチします。この、 `arg1`、 `arg2`、 `kwarg1`または `kwarg2`。 `func(kwarg3=10)`。

- Python 3では、 `*`のみをして、そのすべてのをキーワードとしてするがあることをすることができます。たとえば、Python 3.5 `math.isclose`は、 `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)`をしてされています。つまり、の2つのはできますが、3および4のパラメータはキーワードとしてのみすることができる。

Python 2.xはキーワードのみのパラメータをサポートしていません。このは `kwargs` でエミュレートできます

```
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 ...
```

けにする

オプションのの `args`とオプションのキーワード `kwargs` も、あなたはあなたがきなをすることができるだけのですが、のあなたがをしているかっているように、にうことにである、またはがでそうしてください。

にする

のは、 **none**または1つの `*args`、 **none**または `**kwargs` ですることができますが、のではできません。また、 `*args`のでなければならず、 `**kwargs`のパラメータでなければなりません。いずれかのをしようとする、エラーのがします。

オプションのをつれにする

このようなをネストすることはですが、コードがすでにしたをするのがのですが、パラメータをすは、オプションのに `*`を、オプションのキーワード `args`に `**`をすがありますそうでなければ、 `args`はリストまたはタプルとしてされ、 `kwargs`はのとしてされます。えは

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)

def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

オプションのをしてをする

オプションのをなデフォルトの オプションのをつのをですと、しないにつながるがあります。

このは、のデフォルトががされたでだけされ、がびされたときのくのののようにされ ないためにします。デフォルトは、オブジェクトの__defaults__メンバにされます。

```
def f(a, b=42, c=[]):  
    pass  
  
print(f.__defaults__)  
# Out: (42, [])
```

の のけしとを、をするがないため、これはではありません。のをせずにりてすることができます。したがって、はじデフォルトをつことがされます。しかし、の、さまざまなメンバーをびすことによって、のをすることができます。したがって、へのびしは、のデフォルトをつことがされていません。

```
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]
```

PyCharmのようなIDEでは、ながデフォルトのとしてされているときにがされます

。

デフォルトのは、にあなたがでした1つであることをしたいは、は、にデフォルトのとしてのをすることです。

タイプがデフォルトとしてとされるときにこれをするためのなイディオムは、しないことである None デフォルトとしてを、いでそれがいい、ののにデフォルトをりて None 。

```
def append(elem, to=None):  
    if to is None:  
        to = []  
  
    to.append(elem)  
    return to
```

ラムダインライン

`lambda` キーワードは、のをむインラインをします。こののは、びされたときにがすです。

のをえてみましょう。

```
def greeting():  
    return "Hello"
```

それは、とばれるとき

```
print(greeting())
```

プリントする

```
Hello
```

これはのようにラムダとしてくことができます

```
greet_me = lambda: "Hello"
```

ラムダのへのりてについては、このセクションののをしてください。には、しないでください。

`Hello` をす `greet_me` というのインラインをします。ラムダでをするときには、`return` かないことにしてください。 `after :` のがにされます。

ひとたびにされると、それはののようになことができます

```
print(greet_me())
```

プリントする

```
Hello
```

`lambda` もをることができます

```
strip_and_upper_case = lambda s: s.strip().upper()  
  
strip_and_upper_case(" Hello ")
```

のをします。

```
HELLO
```

ののように、のの/キーワードをすることもできます。

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

プリントする

```
hello ('world',) {'world': 'world'}
```

lambdaは、びされるポイントでするのがないsorted、filter、mapによくされます。

えは、このはのリストをソートしてとをし、とのをします。

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip().upper())
# Out:
# ['    bAR', 'BaZ    ', ' foo ']
```

ソートリストはをするだけです

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip())
# Out:
# ['BaZ    ', '    bAR', ' foo ']
```

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']
```

リストの

```
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]
```

ラムダのからののをびすことができます。

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
```



```
greet()
```

プリントする

```
hello world
```

lambda は1つのだけがまれ、をすることでのステートメントをできるので、これはです。

PEP-8 のPythonスタイルのガイドでは、ラムダをにりてることはおめしませんが2つのでったように

ラムダをにバインドするではなく、にdefをしてください。

はい

```
def f(x): return 2*x
```

いいえ

```
f = lambda x: 2*x
```

のは、オブジェクトのがの<lambda>ではなく f であることをします。これはにトレースバックやのにちます。をすると、ラムダがな def でできるのつまり、よりきなのめむことができるをします。

のけしと

まず、いくつかの

- のパラメータにされるの。
- パラメータ パラメータでされる。

Python では、はによってされます のとはなり、は//ポインタによってされます。

- パラメータをすると、がされますのがな。

```
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
```

- パラメータをりてしても、はりてされません。

```
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]
```

Pythonでは、`y`には`x`を代入するのではなく、`x`となすをオブジェクトにバインドつまりりて、アタッチします。

- `list`、`tuple`など。すべてのでコピーがされます。
- `list`、`set`など。はしてもしなくてもよい。

```
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]
```

Pythonのクロージャは、`id`によってされます。ここで、`makeInc`の`id`は、`inc`でされる`x`バインディングをします。`makeInc`をびすたびにこののしいインスタンスがされますが、インスタンスには`x`なるバインディングへのリンクがあります。

```
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
```

のクロージャでは、`makeInc`はそのむからすべてののをにしますが、このでは、`makeInc`はされたへのみりアクセスのみをちますが、

```
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では、ネストされた変数に対して `nonlocal` [Nonlocal Variables](#) をしています。

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

```

は、そののでびすです。たとえば、`factorial(n) = n*(n-1)*(n-2)*...*3*2*1`でされる、。のよう
にプログラムすることができます

```

def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)

```

ここでののはのとおりです。

```

factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6

```

り。これはであることにしてください。なぜなら、2の`return factorial(n-1)`は、がそののでびす
からです。

いくつかのは[lambda](#)をしてできますが、`lambda`をするはのようになります。

```

factorial = lambda n: 1 if n == 0 else n*factorial(n-1)

```

このはとじをします。

なのさにはがあります。これはPythonのにします。にすると、RuntimeErrorがします。

```
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!
```

して、のさのをすることがであるsys.setrecursionlimit(limit)によってこのをチェック
sys.getrecursionlimit()

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

Python 3.5からの、 RuntimeError からした RecursionError RuntimeError。

ネストされた

Pythonのはファーストクラスのオブジェクトです。それらはのスコープですることができます

```
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
```

みスコープをキャプチャするは、のあらゆるのオブジェクトとにすことができます

```
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
```

な

では、、、の、キーワードargskwargsのようなパラメータのタイプをできます。ここでは、それぞれのタイプのでないがあります。

```
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'
```

きパラメータのをする

シグニチャーののアスタリスクのにされたすべてのパラメーターはキーワードのみです。

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

Python 3では、のシグネチャにアスタリスクを1つれてりのをキーワードですことができます。

```
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
```

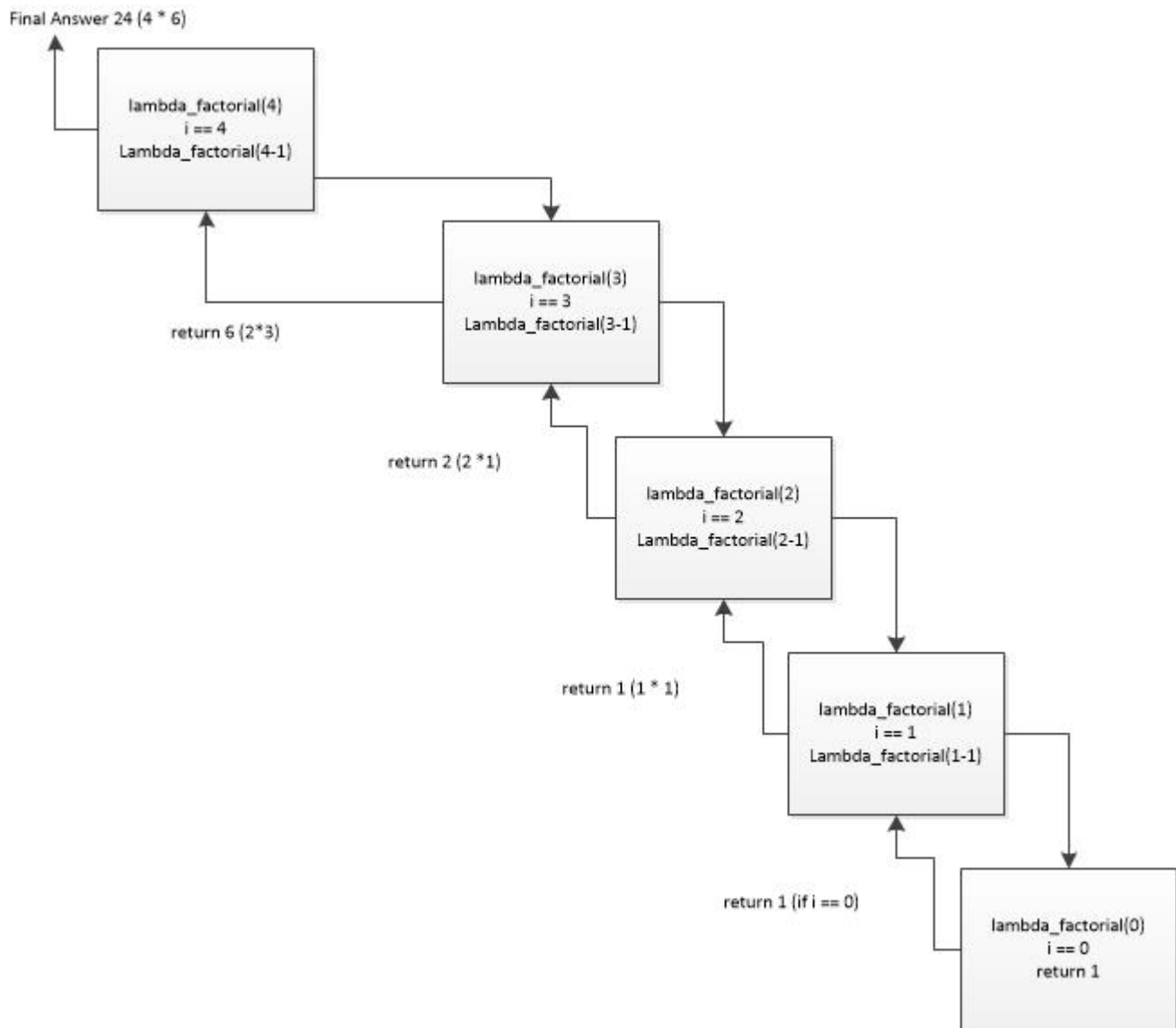
りてられたをするラムダ

ラムダをする1つのは、にをし、のでそのをすることです。これのなは、のコードにすように、のをにすることです。

```
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
```

コードの

ラムダは、そののりてをして、それが0であるか、またはそうでなければのした、それは1をし、り4をさ_i *のラムダによっての-1 _{i-1}。これは、されたが0 `return 1` にデクリメントされるまでできます。のようにできるプロセス



オンラインでもむ <https://riptutorial.com/ja/python/topic/228/>

205: でのループの

き

Pythonでは、が "return"ステートメントにするとすぐにされます。

Examples

のループのをす

このでは、varが1になるとすぐにがります

```
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])
```

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>>> Got 1
```

オンラインででのループのをむ <https://riptutorial.com/ja/python/topic/10883/でのループの>

Examples

のオーバーロード

Pythonのすべてがオブジェクトです。オブジェクトには、のオブジェクトとやりとりするためにするなメソッドがあります。に、これらのメソッドは__action__にあります。に、これはPythonデータモデルとされます。

これらののいずれかをオーバーロードすることができます。これはPythonのオーバーロードでよくされます。は、Pythonのデータモデルをしたのオーバーロードのです。Vectorクラスは、2つのなベクトルをします。のオーバーロードをして2つのベクトルのになサポートをします。

```
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)>
```

のは、なのオーバーロードをしています。なリストは[ここ](#)でつけることができます。

オンラインでしをむ <https://riptutorial.com/ja/python/topic/946/> し

S. No		Contributors
1	Pythonをいめる	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, gIS, GoatsWearHats, GThamizh, H. Pauwelyn, hardmooth, hayalci, hichris123, Ian, IanAuld, icesin, Igor Raush, Ilyas Mimouni, itshejoker, J F, Jabba, jalanb, James, James Taylor, Jean-Francois T., jedwards, Jeffrey Lin, jfunez, JGreenwell, Jim Fasarakis Hilliard, jim opleyduIVEN, 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, Matthew 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, nouλλΔ, ÁzεJQ, 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, Shadowfa, 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	"pip"モジュールのい PyPI Package Manager	Zydnar
3	* argsと** kwargs	cjds , Eric Zhang , ericmarkmartin , Geeklhern , J F , Jeff Hutchins , Jim Fasarakis Hilliard , JuanPablo , kdopen , loading... , Marlon Abeykoon , Matthew Whitt , Pasha , pcurry , PsyKzz , Scott Mermelstein , user2314737 , Valentin Lorentz , Veedrac
4	__name__	Anonymous , BusyAnt , Christian Ternes , jonrsharpe , Lutz Prechelt , Steven Elliott
5	`exec`と`eval`をつた コード	Antti Haapala , Ilja Everilä
6	2to3ツール	Alessandro Trinca Tornidor , Dartmouth , Firix , Kevin Brown , Naga2Raja , Stephen Leppik
7	AMQPStormをした RabbitMQの	eandersson
8	ArcPy	Midavalo , PolyGeo , Zhanping Shi
9	Asyncioモジュール	2Cubed , Alessandro Trinca Tornidor , Cimbali , hiro protagonist , obust , pylang , RamenChef , Seth M. Larson , Simplans , Stephen Leppik , Udi
10	base64モジュール	Thomas Gerot
11	ChemPy - pythonパ ッケージ	Biswa_9937
12	configparser	Chinmay Hegde , Dunatotatos
13	CSVのみき	Adam Matan , Franck DERNONCOURT , Martin Valgur , mnoronha , ravigadila , Setu
14	ctypes	Or East
15	Dequeモジュール	Anthony Pham , BusyAnt , matsjoyce , ravigadila , Simplans , Thomas Ahle , user2314737
16	Functoolsモジュール	Alessandro Trinca Tornidor , enrico.bacis , flamenco , RamenChef , Shrey Gupta , Simplans , Stephen Leppik , StuxCrystal

17	groupby	Parousia , Thomas Gerot
18	GZipでめる	orvi
19	HTML	alecxe , talhasch
20	ijson	Prem Narain
21	Itertoolsモジュール	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
22	JSONモジュール	Indradhanush Gupta , Leo , Martijn Pieters , pzp , theheadofabroom , Underyx , Wolfgang
23	kivy - NUIのためのクロスプラットフォームPythonフレームワーク	dhimanta
24	Matplotlibでプロットする	Arun , user2314737
25	MutableとPythonのおよびHashable	Cilyan
26	os.path	Claudiu , Fábio Perez , girish946 , Jmills , Szabolcs Dombi , VJ Magar
27	osモジュール	Andy , Christian Ternes , JelmerS , JL Peyret , mnoronha , Vinzee
28	Pandas Transformグループのをプリフォームし、をする	Dee
29	Pandasをしたデータファイルの、サブセットおよび	Mark Miller
30	pipPyPIパッケージマネージャ	Andy , Arpit Solanki , Community , InitializeSahib , JNat , Mahdi , Majid , Matt Giltaji , Nathaniel Ford , Rápli András , SerialDev , Simplans , Steve Barnes , StuxCrystal , tlo
31	PostgreSQL	Alessandro Trinca Tornidor , RamenChef , Stephen Leppik , user2027202827
32	py.test	Andy , Claudiu , Ffisegydd , Kinifwyne , Matt Giltaji

33	Py2Neoを使ったNeo4jとCypher	Wingston Sharon
34	pyautoguiモジュール	Damien, Rednivrug
35	PyInstaller - Pythonコードの	ChaoticTwist, Eric, mnoronha
36	Python 2からPython 3への	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, CLDSEED, Colin Yang, Cometsong, Community, Conrad.Dean, danidee, Daniel Stradowski, Darth Shadow, Dartmouth, Dave J, David Cullen, David Heyman, deenes, 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, Matthew Whitt, Max Feng, mgwilliams, Michael Recachinas, mkj, mnoronha, Moinuddin Quadri, muddyfish, Nathaniel Ford, niemmi, niyasc, nouʎɹdʌʒɐɹO, OrangeTux, Pasha, Paul Weaver, Paulo Freitas, pcurry, pktangyue, poppie, pylang, python273, Pythonista, RahulHP, Rakitić, RamenChef, Rauf, René G, rfkortekaas, rrao, Ryan, sbclair, 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
37	Python HTTP Server	Arpit Solanki, J F, jmunsch, Justin Chadwell, Mark, MervS, orvi , quantummind, Raghav, RamenChef, Sachin Kalkur, Simplans , techydesigner

38	Python Lex-Yacc	CLDSEED
39	Python Server Sent イベント	Nick Humrich
40	Pythonシリアル pyserial	Alessandro Trinca Tornidor , Ani Menon , girish946 , mnoronha , Saranjith , user2314737
41	Pythonデータ	Gavin , lorenzofeliz , Pike D. , Rednivrug
42	Pythonでのセキュア シェア	mnoronha , Shijo
43	Pythonでのない	4444 , Guy , kollery , Vinzee
44	Pythonでのプログラ ミング	Imran Bughio , mvis89 , Rednivrug
45	PythonとExcel	bee-sting , Chinmay Hegde , GiantsLoveDeathMetal , hackvan , Majid , talhasch , user2314737 , Will
46	PythonとRaspberry PIをしたIoTプログラ ミング	dhimanta
47	PythonとSQL Server の	metmirr
48	PythonによるWebス クレイピング	alecxe , Amitay Stern , jmunsch , mrtuovinen , Ni. , RamenChef , Saiful Azad , Saqib Shamsi , Simplans , Steven Maude , sth , sytech , talhasch , Thomas Gerot
49	Pythonによるデータ	Aquib Javed Khan , Arun , ChaoticTwist , cledoux , Ffisegydd , ifma
50	Pythonネットワーク	atayenel , ChaoticTwist , David , Geeklhern , mattgathu , mnoronha , thsecmaniac
51	Pythonのアンチパタ ーン	Alessandro Trinca Tornidor , Anonymous , eenblam , Mahmoud Hashemi , RamenChef , Stephen Leppik
52	Pythonのソースコー ドとバイトコードへ のアクセス	muddyfish , StuxCrystal , user2314737
53	Pythonのテンプレー ト	4444 , Alessandro Trinca Tornidor , Fred Barclay , RamenChef , Ricardo , Stephen Leppik

54	Pythonの	David Heyman , Faiz Halde , Iván Rodríguez Torres , J F , Thomas Moreau , Tyler Gubala
55	Pythonの	RamenChef , user2728397
56	Pythonパッケージの	Claudiu , KeyWeeUsr , Marco Pashkov , pylang , SuperBiasedMan , Thtu
57	Pythonリクエスト Post	Ken Y-N , RandomHash
58	PythonをCからびす	Julij Jegorov
59	Pythonをした Windowsサービスの	Simon Hibbs
60	Python - virtualenv	Vikash Kumar Jain
61	setup.py	Adam Brenecki , amblina , JNat , ravigadila , strpeter , user2027202827 , Y0da
62	Sqlite3モジュール	Chinmay Hegde , Simplans
63	sys	blubberdiblub
64	tempfile NamedTemporaryFile	Alessandro Trinca Tornidor , amblina , Kevin Brown , Stephen Leppik
65	Unicode	wim
66	Unicodeとバイト	Claudiu , KeyWeeUsr
67	virtualenvwrapperを った	Sirajus Salayhin
68	Websockets	2Cubed , Stephen Leppik , Tyler Gubala
69	Webサーバーゲート ウェイインターフェ イスWSGI	David Heyman , Kevin Brown , Preston , techydesigner
70	Webブラウザモジュ ール	Thomas Gerot
71	Windowsで virtualenvwrapperを ってをする	Sirajus Salayhin
72	XMLの	4444 , Brad Larson , Chinmay Hegde , Francisco Guimaraes ,

greuze, heyhey2k, Rob Murray		
73	ZIPアーカイブをとって する	Chinmay Hegde, ghostarbeiter, Jeffrey Lin, SuperBiasedMan
74	イテレータとイテレ ータ	4444, Conrad.Dean, demonplus, Ilia Barahovski, Pythonista
75	インタプリタコマン ドラインコンソール	Aaron Christiansen, David, Elazar, Peter Shinnars, ppperry
76	インデックスとスラ イス	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
77	インデント	Alessandro Trinca Tornidor, depperm, J F, JGreenwell, Matt Giltaji, Pasha, RamenChef, Stephen Leppik
78	ウルリブ	Amitay Stern, ravigadila, sth, Will
79	オーディオ	blueberryfields, Comrade SparklePony, frankyjuang, jmunsch, orvi, qwertyuip9, Stephen Leppik, Thomas Gerot
80	オーバーロード	Andy Hayden, Darth Shadow, ericmarkmartin, Ffisegydd, Igor Raush, Jonas S, jonrsharpe, L3viathan, Majid, RamenChef, Simplans, Valentin Lorentz
81	オペレータの	HoverHell, JGreenwell, MathSquared, SashaZd, Shreyash S Sarnayak
82	カウント	Andy Hayden, MSeifert, Peter Mølgaard Pallesen, pylang
83	カスタムエラー/をさ せる	naren
84	ガベージコレクション	bogdanciobanu, Claudiu, Conrad.Dean, Elazar, FazeL, J F, James Elderfield, lukess, muddyfish, Sam Whited, Siggys, Stephen Leppik, SuperBiasedMan, Xavier Combelle
85	キューモジュール	Prem Narain
86	クライアントとサー バーのソケットとメ ッセージの/	Mohammad Julfikar
87	クラス	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, Imiguelvargasf, Mailerdaimon, Martijn Pieters, Massimiliano Kraus, Matthew 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, wrrwr, Yogendra Sharma
88	クラスインスタンスの__str__および__repr__メソッド	Alessandro Trinca Tornidor, jedwards, JelmerS, RamenChef, Stephen Leppik
89	グラフツール	xiaoyi
90	グローバルインタープリタロックGILをする	Scott Mermelstein
91	コードブロック、フレーム、および	Jeremy, Mohammed Salman
92	コマンドラインの	amblina, Braiam, Claudiu, cledoux, Elazar, Gerard Roche, krato, loading..., Marco Pashkov, Or Duan, Pasha, RamenChef, rfkortekaas, Simplans, Thomas Gerot, Topperfalkon, zmo, zondo
93	コメントとドキュメント	Ani Menon, FunkySayu, MattCorr, SuperBiasedMan, TuringTux
94	コレクションモジュール	asmeurer, Community, Elazar, jmunsch, kon psych, Marco Pashkov, MSeifert, RamenChef, Shawn Mehan, Simplans, Steven Maude, Symmitchry, void, XCoder Real
95	コンテキストマネージャ "with"ステートメント	Abhijeet Kasurde, Alessandro Trinca Tornidor, Andy Hayden, Antoine Bolvy, carrdelling, Conrad.Dean, Dartmouth, David 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
96	サブプロセスライブラリ	Adam Matan, Andrew Schade, Brendan Abel, jfs, jmunsch, Riccardo Petraglia
97	ジャンゴ	code_geek, orvi

		tyteen4a03 , Wladimir Palant , zvone
112	デザインパターン	Charul , denvaar , djaszczurowski
113	デバッグ	Aldo , B8vrede , joel3000 , Sardathrion , Sardorbek Imomaliev , Vlad Bezden
114	トキンター	Dartmouth , rlee827 , Thomas Gerot , TidB
115	パイオディオ	Biswa_9937
116	パイゲーム	Anthony Pham , Aryaman Arora , Pavan Nath
117	バイナリデータ	Eleftheria , evuez , mnoronha
118	パスステートメント	Anaphory
119	パスのとの	Esteis , Marlon Abeykoon , mnoronha , PYPL
120	ハッシュライブラリ	Mark Omo , xiaoyi
121	パフォーマンスの	A. Ciclet , RamenChef , user2314737
122	ヒーブック	ettanany
123	ピクルデータのシリアライゼーション	J F , Majid , Or East , RahulHP , rfkortekaas , zvone
124	ピグレット	Comrade SparklePony , Stephen Leppik
125	ビット	Abhishek Jain , boboquack , Charles , Gal Dreiman , intboolstring , JakeD , JNat , Kevin Brown , Matías Brignone , nemesifixx , poke , R Colmenares , Shawn Mehan , Simplans , Thomas Gerot , tmr232 , Tony Suffolk 66 , viveksyng
126	ファイルとフォルダのI/O	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
127	ファイルの	andrew
128	フィルタ	APerson , cfi , J Atkin , MSeifert , rajah9 , SuperBiasedMan
129	ブール	boboquack , Brett Cannon , Dair , Ffisegydd , John Zwinck , Severiano Jaramillo Quintanar , Steven Maude

130	プラグインとエクステンションクラス	2Cubed , proprefenetre , pylang , rrao , Simon Hibbs , Simplans
131	フラスコ	Stephen Leppik , Thomas Gerot
132	プログラムのPython	ADITYA , Antonio , Elodin , Neil A. , Vinzee
133	プロセスとスレッド	Claudiu , Thomas Gerot
134	プロパティオブジェクト	Alessandro Trinca Tornidor , Darth Shadow , DhiaTN , J F , Jacques de Hooge , Leo , Martijn Pieters , mnoronha , Priya , RamenChef , Stephen Leppik
135	プロファイリング	J F , keiv.fly , SashaZd
136	マルチスレッド	Alu , CLDSEED , juggernaut , Kevin Brown , Kristof , mattgathu , Nabeel Ahmed , nlsdfnbch , Rahul , Rahul Nair , Riccardo Petraglia , Thomas Gerot , Will , Yogendra Sharma
137	マルチプロセッシング	Alon Alexander , Nander Speerstra , unutbu , Vinzee , Will
138	ミックスイン	Doc , Rahul Nair , SashaZd
139	メソッドオーバーライド	DeepSpace , James
140	メタクラス	2Cubed , Amir Rachum , Antoine Pinsard , Camsbury , Community , driax , Igor Raush , InitializeSahib , Marco Pashkov , Martijn Pieters , Matthew Whitt , OozeMeister , Pasha , Paulo Scardine , RamenChef , Rob Bednark , Simplans , sisanared , zvone
141	モジュールとパッケージのい	DeepSpace , Simplans , tjohnson
142	モジュールのインポート	angussidney , Anthony Pham , Antonis Kalou , Brett Cannon , BusyAnt , Casebash , Christian Ternes , 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
143	ユーザーメソッド	Alessandro Trinca Tornidor , Beall619 , mnoronha , RamenChef ,

		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 , JOHN , 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 , Kwartz , Lafexlos , lapis , Lee Netherton , Liteye , Locane , Lyndsy Simon , machine yearning , Mahdi , Marc , Markus Meskanen , Martijn Pieters , Matt , Matt Giltaji , Matt S , Matthew Whitt , Maximillian Laumeister , mbrig , Mirec Miskuf , Mitch Talmadge , Morgan Thrapp , MSeifert , muddyfish , n8henrie , Nathan Arthur , nehemiah , nouλdλzελC , 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
150	リストによるの	zenlc2000
151	リストパッキングとアンパックともいう	J F , sth , zmo
152	リンクされたリスト	Nemo
153	リンクされたリストノード	orvi
154	ループ	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, skrrgwasmе, SuperBiasedMan, textshell, The_Curry_Man, Thomas Gerot, Tom, Tony Suffolk 66, user1349663, user2314737, Vinzee, Will
155	ロギング	Gal Dreiman, Jörn Hees, sxnwlfkk
156	ロケールモジュール	Will, XonAether
157	べえ、と	Antti Haapala, APerson, GoatsWearHats, Mirec Miskuf, MSeifert, RamenChef, Simplans, Valentin Lorentz
158		Akshat Mahajan, Dair, Franck Dernoncourt, J F, Mahdi, nlsdfnbch, Ryan Smith, Vinzee, Xavier Combelle
159		Adrian17, Artem Kolontay, ArtOfCode, Bhargav, brennan, Dair, Daniil Ryzhkov, Darkade, Darth Shadow, edwinksl, Fernando, ghostarbeiter, ha_1694, Hans Then, lancnorden, J F, Majid, Marco Pashkov, Matt Giltaji, Matthew 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
160		Adrian Antunez, Alessandro Trinca Tornidor, Alfe, Andy, Benjamin Hodgson, Brian Rodriguez, BusyAnt, Claudiu, driax, Elazar, flazzarini, ghostarbeiter, Ilia Barahovski, J F, Marco Pashkov, muddyfish, noukAdAzeJO, Paul Weaver, Rahul Nair, RamenChef, Shawn Mehan, Shiven, Shkelqim Memolla, Simplans, Slickytail, Stephen Leppik, Sudip Bhandari, SuperBiasedMan, user2314737
161		rassar
162	でないPythonの	Jacques de Hooge, Squidward
163	とし	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
164		Bastian, japborst, JGreenwell, Jossie Calderon, mbomb007, SashaZd, Tyler Crompton

165		Alessandro Trinca Tornidor , JGreenwell , metahost , Pigman168 , RamenChef , Stephen Leppik
166		Andy , Elazar , evuez , Martijn Pieters , techydesigner
167	な	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
168		Beall619 , Frustrated , Justin , Leon Z. , lukewrites , SuperBiasedMan , Valentin Lorentz
169	スコープとバインディング	Anthony Pham , davidism , Elazar , Esteis , Mike Driscoll , SuperBiasedMan , user2314737 , zvone
170		APerson , cfi , Igor Raush , Jon Ericson , Karl Knechtel , Marco Pashkov , MSeifert , nouλdλzελQ , Parousia , Simplans , SuperBiasedMan , tlama , user2314737
171		Doraemon , GoatsWearHats , J F , JNat , Marco Pashkov , Mark Miller , Martijn Pieters , Nathaniel Ford , Nicolás , pcurry , pzp , SashaZd , SuperBiasedMan , Vilmar
172	なデータint、float、str、tuple、frozensets	Alessandro Trinca Tornidor , FazeL , Ganesh K , RamenChef , Stephen Leppik
173		Benedict Bunting , DeepSpace , depperm , Simplans , skrrgwasme , Vinzee
174		boboquack , Buzz , rrao
175	アクセス	Elazar , SashaZd , SuperBiasedMan
176		Benjamin Hodgson , Elazar , Faiz Halde , J F , Lee Netherton , loading... , Mister Mister
177	クラスabc	Akshat Mahajan , Alessandro Trinca Tornidor , JGreenwell , Kevin Brown , Matthew Whitt , mkrieger1 , SashaZd , Stephen Leppik
178		Teepeemm
179	の	Dartmouth , J F , mattgathu , Nathan Osman , techydesigner , ygram
180	モジュール	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
181	の	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
182	またはリストからの CSVへのきみ	Hriddhi Dey, Thomas Crowley
183	メソッド	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
184	の	surfthecity
185		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
186		Andy Hayden, BusyAnt, Chris Larson, deepakkt, Delgan, Elazar, evuez, Ffisegydd, Geeklhem, Hannes Karppila, James, Kevin Brown, krato, Max Feng, nou!Kd!ze!O, rajah9, rrao, SashaZd, Simplans, Slayther, Soumendra Kumar Sahoo, Thomas Gerot, Trimax, Valentin Lorentz, Vinzee, wwii, xgord, Zack
187		Razik

188		Biswa_9937
189		Dan Sanderson , Igor Raush , MSeifert
190	の、のいPythonとJavaScript	user2683246
191	なヘルプをつCLIサブコマンド	Alessandro Trinca Tornidor , anatoly techtonik , Darth Shadow
192		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
193		Anthony Pham , Ares , Elazar , J F , MSeifert , Shawn Mehan , SuperBiasedMan , Will , Xavier Combelle
194	らす	APerson , Igor Raush , Martijn Pieters , MSeifert
195	モジュール	MSeifert
196		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 , JOHN , 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
197		Anthony Pham , intboolstring , jtbandes , Luke Taylor , MSeifert , Pasha , supersam654
198	な	Adeel Ansari , Bosoneando , bpachev
199		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 , nouλλλzελQ , ronrest , Roy Iacob , Shawn

