EBook Gratuito

# APPRENDIMENTO
# Ruby Language

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#ruby

# Sommario

# Di

# Capitolo 1: Iniziare con Ruby Language

## Osservazioni

Ruby è un linguaggio interpretato multi-piattaforma open source, dinamico orientato agli oggetti, progettato per essere semplicistico e produttivo. È stato creato da Yukihiro Matsumoto (Matz) nel 1995.

> Secondo il suo creatore, Ruby è stato influenzato da Perl , Smalltalk , Eiffel , Ada e Lisp . Supporta diversi paradigmi di programmazione, inclusi funzionale, orientato agli oggetti e imperativo. Ha anche un sistema di tipo dinamico e gestione automatica della memoria.

## Versioni

| Versione | Data di rilascio |
|---|---|
| 2.4 | 2016/12/25 |
| 2.3 | 2015/12/25 |
| 2.2 | 2014/12/25 |
| 2.1 | 2013/12/25 |
| 2.0 | 2013/02/24 |
| 1.9 | 2007-12-25 |
| 1.8 | 2003/08/04 |
| 1.6.8 | 2002/12/24 |

## Examples

### Ciao mondo da IRB

In alternativa, è possibile utilizzare Interactive Ruby Shell (IRB) per eseguire immediatamente le istruzioni Ruby precedentemente scritte nel file Ruby.

Avvia una sessione IRB digitando:

```
$ irb
```

Quindi immettere il seguente comando:

```
puts "Hello World"
```

Ciò risulta nel seguente output della console (incluso newline):

```
Hello World
```

Se non vuoi iniziare una nuova riga, puoi usare la `print` :

```
print "Hello World"
```

### Ciao mondo con tk

Tk è l'interfaccia utente grafica standard (GUI) per Ruby. Fornisce una GUI multipiattaforma per i programmi Ruby.

## Codice di esempio:

```
require "tk"
TkRoot.new{ title "Hello World!" }
Tk.mainloop
```

**Il risultato:**


**Spiegazione passo passo:**

```
require "tk"
```

Carica il pacchetto tk.

```
TkRoot.new{ title "Hello World!" }
```

Definire un widget con il titolo `Hello World`

```
Tk.mainloop
```

Avvia il ciclo principale e visualizza il widget.

## Ciao mondo

> Questo esempio presuppone che Ruby sia installato.

Inserire quanto segue in un file denominato `hello.rb` :

```
puts 'Hello World'
```

Dalla riga di comando, digitare il comando seguente per eseguire il codice Ruby dal file di origine:

```
$ ruby hello.rb
```

Questo dovrebbe produrre:

```
Hello World
```

L'output verrà immediatamente visualizzato sulla console. I file sorgente di Ruby non devono essere compilati prima di essere eseguiti. L'interprete Ruby compila ed esegue il file Ruby in fase di runtime.

## Ciao mondo senza file di origine

Esegui il comando seguente in una shell dopo aver installato Ruby. Questo mostra come è possibile eseguire semplici programmi Ruby senza creare un file Ruby:

```
ruby -e 'puts "Hello World"'
```

È anche possibile alimentare un programma Ruby all'input standard dell'interprete. Un modo per farlo è usare un documento qui nel comando shell:

```
ruby <<END
puts "Hello World"
END
```

## Hello World come un file eseguibile da soli con l'uso di Shebang (solo sistemi operativi simili a Unix)

È possibile aggiungere una direttiva interprete (shebang) al proprio script. Crea un file chiamato `hello_world.rb` che contiene:

```
#!/usr/bin/env ruby

puts 'Hello World!'
```

Fornisci le autorizzazioni eseguibili dello script. Ecco come farlo in Unix:

```
$ chmod u+x hello_world.rb
```

Ora non è necessario chiamare l'interprete Ruby esplicitamente per eseguire il tuo script.

```
$ ./hello_world.rb
```

### Il mio primo metodo

# Panoramica

Crea un nuovo file chiamato `my_first_method.rb`

Inserisci il seguente codice all'interno del file:

```
def hello_world
  puts "Hello world!"
end

hello_world() # or just 'hello_world' (without parenthesis)
```

Ora, da una riga di comando, eseguire quanto segue:

```
ruby my_first_method.rb
```

L'output dovrebbe essere:

```
Hello world!
```

# Spiegazione

- `def` è una parola chiave che ci dice che stiamo `def` -ining un metodo - in questo caso, `hello_world` è il nome del nostro metodo.
- `puts "Hello world!"` `puts` (o pipe alla console) la stringa `Hello world!`
- `end` è una parola chiave che significa che stiamo finendo la nostra definizione del metodo `hello_world`
- poiché il metodo `hello_world` non accetta argomenti, puoi omettere la parentesi richiamando il metodo

Leggi Iniziare con Ruby Language online: https://riptutorial.com/it/ruby/topic/195/iniziare-con-ruby-language

# Capitolo 2: Ambito e visibilità variabili

## Sintassi

- $ global_variable
- @@ class_variable
- @instance_variable
- local_variable

## Osservazioni

Le variabili di classe sono condivise nella gerarchia di classi. Ciò può comportare un comportamento sorprendente.

```
class A
  @@variable = :x

  def self.variable
    @@variable
  end
end

class B < A
  @@variable = :y
end

A.variable  # :y
```

Le classi sono oggetti, quindi è possibile utilizzare variabili di istanza per fornire lo stato specifico per ogni classe.

```
class A
  @variable = :x

  def self.variable
    @variable
  end
end

class B < A
  @variable = :y
end

A.variable  # :x
```

## Examples

### Variabili locali

Le variabili locali (a differenza delle altre classi variabili) non hanno alcun prefisso

```
local_variable = "local"
p local_variable
# => local
```

Il suo ambito dipende da dove è stato dichiarato, non può essere usato al di fuori dell'ambito "contenitori di dichiarazione". Ad esempio, se una variabile locale è dichiarata in un metodo, può essere utilizzata solo all'interno di quel metodo.

```
def some_method
    method_scope_var = "hi there"
    p method_scope_var
end

some_method
# hi there
# => hi there

method_scope_var
# NameError: undefined local variable or method `method_scope_var'
```

Naturalmente, le variabili locali non sono limitate ai metodi, come una regola empirica si potrebbe dire che, non appena si dichiara una variabile all'interno di un `do ... end` blocco `do ... end` o racchiusa tra parentesi graffe `{}` , sarà locale e con scope il blocco è stato dichiarato in

```
2.times do |n|
    local_var = n + 1
    p local_var
end
# 1
# 2
# => 2

local_var
# NameError: undefined local variable or method `local_var'
```

Tuttavia, le variabili locali dichiarate in `if` o `case` block possono essere utilizzate nello scope parent:

```
if true
    usable = "yay"
end

p usable
# yay
# => "yay"
```

Mentre le variabili locali non possono essere utilizzate al di fuori del suo blocco di dichiarazione, verranno passate ai blocchi:

```
my_variable = "foo"
```

```
my_variable.split("").each_with_index do |char, i|
    puts "The character in string '#{my_variable}' at index #{i} is #{char}"
end
# The character in string 'foo' at index 0 is f
# The character in string 'foo' at index 1 is o
# The character in string 'foo' at index 2 is o
# => ["f", "o", "o"]
```

Ma non alle definizioni metodo / classe / modulo

```
my_variable = "foo"

def some_method
    puts "you can't use the local variable in here, see? #{my_variable}"
end

some_method
# NameError: undefined local variable or method `my_variable'
```

Le variabili utilizzate per gli argomenti di blocco sono (ovviamente) locali al blocco, ma oscureranno le variabili precedentemente definite, senza sovrascriverle.

```
overshadowed = "sunlight"

["darkness"].each do |overshadowed|
    p overshadowed
end
# darkness
# => ["darkness"]

p overshadowed
# "sunlight"
# => "sunlight"
```

## Variabili di classe

Le variabili di classe hanno un ambito di classe, possono essere dichiarate ovunque nella classe. Una variabile sarà considerata una variabile di classe quando è preceduta da `@@`

```
class Dinosaur
    @@classification = "Like a Reptile, but like a bird"

    def self.classification
        @@classification
    end

    def classification
        @@classification
    end
end

dino = Dinosaur.new
dino.classification
# => "Like a Reptile, but like a bird"

Dinosaur.classification
```

```
# => "Like a Reptile, but like a bird"
```

Le variabili di classe sono condivise tra classi correlate e possono essere sovrascritte da una classe figlia

```
class TRex < Dinosaur
    @@classification = "Big teeth bird!"
end

TRex.classification
# => "Big teeth bird!"

Dinosaur.classification
# => "Big teeth bird!"
```

Questo comportamento è indesiderato per la maggior parte del tempo e può essere aggirato utilizzando variabili di istanza a livello di classe.

Le variabili di classe definite all'interno di un modulo non sovrascriveranno le loro variabili di classe comprese le classi:

```
module SomethingStrange
    @@classification = "Something Strange"
end

class DuckDinosaur < Dinosaur
    include SomethingStrange
end

DuckDinosaur.class_variables
# => [:@@classification]
SomethingStrange.class_variables
# => [:@@classification]

DuckDinosaur.classification
# => "Big teeth bird!"
```

## Variabili globali

Le variabili globali hanno una portata globale e, quindi, possono essere utilizzate ovunque. Il loro scopo non dipende da dove sono definiti. Una variabile sarà considerata globale, se preceduta da un segno `$` .

```
$i_am_global = "omg"

class Dinosaur
    def instance_method
       p "global vars can be used everywhere. See? #{$i_am_global}, #{$another_global_var}"
    end

    def self.class_method
       $another_global_var = "srsly?"
       p "global vars can be used everywhere. See? #{$i_am_global}"
    end
```

```
end

Dinosaur.class_method
# "global vars can be used everywhere. See? omg"
# => "global vars can be used everywhere. See? omg"

dinosaur = Dinosaur.new
dinosaur.instance_method
# "global vars can be used everywhere. See? omg, srsly?"
# => "global vars can be used everywhere. See? omg, srsly?"
```

Poiché una variabile globale può essere definita ovunque e sarà visibile ovunque, la chiamata a una variabile globale "non definita" restituirà nil anziché generare un errore.

```
p $undefined_var
# nil
# => nil
```

Sebbene le variabili globali siano facili da usare, il suo utilizzo è fortemente scoraggiato a favore delle costanti.

### Variabili di istanza

Le variabili di istanza hanno un ambito a livello di oggetto, possono essere dichiarate ovunque nell'oggetto, tuttavia una variabile di istanza dichiarata a livello di classe, sarà visibile solo nell'oggetto classe. Una variabile sarà considerata una variabile di istanza quando è preceduta da `@` . Le variabili di istanza vengono utilizzate per impostare e ottenere attributi di oggetti e restituiranno zero se non definite.

```
class Dinosaur
    @base_sound = "rawrr"

    def initialize(sound = nil)
        @sound = sound || self.class.base_sound
    end

    def speak
        @sound
    end

    def try_to_speak
        @base_sound
    end

    def count_and_store_sound_length
        @sound.chars.each_with_index do |char, i|
            @sound_length = i + 1
            p "#{char}: #{sound_length}"
        end
    end

    def sound_length
        @sound_length
    end
```

```
    def self.base_sound
        @base_sound
    end
end

dino_1 = Dinosaur.new
dino_2 = Dinosaur.new "grrr"

Dinosaur.base_sound
# => "rawrr"
dino_2.speak
# => "grrr"
```

Non è possibile accedere alla variabile di istanza dichiarata a livello di classe a livello di oggetto:

```
dino_1.try_to_speak
# => nil
```

Tuttavia, abbiamo usato la variabile di istanza `@base_sound` per istanziare il suono quando non viene passato alcun suono al nuovo metodo:

```
dino_1.speak
# => "rawwr"
```

Le variabili di istanza possono essere dichiarate ovunque nell'oggetto, anche all'interno di un blocco:

```
dino_1.count_and_store_sound_length
# "r: 1"
# "a: 2"
# "w: 3"
# "r: 4"
# "r: 5"
# => ["r", "a", "w", "r", "r"]

dino_1.sound_length
# => 5
```

Le variabili di istanza **non** sono condivise tra istanze della stessa classe

```
dino_2.sound_length
# => nil
```

Questo può essere usato per creare variabili di livello di classe, che non verranno sovrascritte da una classe figlio, poiché le classi sono anche oggetti in Ruby.

```
class DuckDuckDinosaur < Dinosaur
    @base_sound = "quack quack"
end

duck_dino = DuckDuckDinosaur.new
duck_dino.speak
# => "quack quack"
DuckDuckDinosaur.base_sound
```

```
# => "quack quack"
Dinosaur.base_sound
# => "rawrr"
```

Leggi Ambito e visibilità variabili online: https://riptutorial.com/it/ruby/topic/4094/ambito-e-visibilita-variabili

# Capitolo 3: Applicazioni a riga di comando

## Examples

### Come scrivere uno strumento da riga di comando per ottenere il meteo tramite codice di avviamento postale

Questo sarà un tutorial relativamente completo su come scrivere uno strumento da riga di comando per stampare il tempo dal codice postale fornito allo strumento da riga di comando. Il primo passo è scrivere il programma in ruby per fare questa azione. Iniziamo scrivendo un metodo `weather(zip_code)` (Questo metodo richiede la gemma `yahoo_weatherman` . Se non hai questa gemma puoi installarla digitando `gem install yahoo_weatherman` dalla riga di comando)

```
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end
```

Ora abbiamo un metodo molto semplice che fornisce il tempo quando viene fornito un codice postale. Ora dobbiamo renderlo uno strumento da riga di comando. Molto rapidamente esaminiamo come viene chiamato uno strumento da riga di comando dalla shell e le variabili associate. Quando uno strumento è chiamato come questo `tool argument other_argument` , in ruby c'è una variabile `ARGV` che è una matrice uguale a `['argument', 'other_argument']` . Ora implementiamo questo nella nostra applicazione

```
#!/usr/bin/ruby
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end

puts weather(ARGV[0])
```

Buono! Ora abbiamo un'applicazione a riga di comando che può essere eseguita. Si noti la linea *she-bang* all'inizio del file ( `#!/usr/bin/ruby` ). Ciò consente al file di diventare un eseguibile. Possiamo salvare questo file come `weather` . ( **Nota** : non salvare questo come `weather.rb` , non è necessario per l'estensione del file e la she-bang dice tutto ciò che è necessario per dire che questo è un file rubino). Ora possiamo eseguire questi comandi nella shell (non digitare `$` ).

```
$ chmod a+x weather
$ ./weather [ZIPCODE]
```

Dopo aver verificato che questo funziona, possiamo ora collegarlo simbolicamente a `/usr/bin/local/` eseguendo questo comando

```
$ sudo ln -s weather /usr/local/bin/weather
```

Ora il `weather` può essere chiamato sulla riga di comando indipendentemente dalla directory in cui ci si trova.

Leggi Applicazioni a riga di comando online: https://riptutorial.com/it/ruby/topic/7679/applicazioni-a-riga-di-comando

# Capitolo 4: Appuntamento

## Sintassi

- DateTime.new (anno, mese, giorno, ora, minuto, secondo)

## Osservazioni

Prima di utilizzare DateTime è necessario `require 'date'`

## Examples

### DateTime da stringa

`DateTime.parse` è un metodo molto utile che costruisce un DateTime da una stringa, indovinando il suo formato.

```
DateTime.parse('Jun, 8 2016')
# => #<DateTime: 2016-06-08T00:00:00+00:00 ((2457548j,0s,0n),+0s,2299161j)>
DateTime.parse('201603082330')
# => #<DateTime: 2016-03-08T23:30:00+00:00 ((2457456j,84600s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50')
# => #<DateTime: 2016-11-04T03:50:00+00:00 ((2457697j,13800s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50 -0300')
# => #<DateTime: 2016-11-04T03:50:00-03:00 ((2457697j,24600s,0n),-10800s,2299161j)
```

Nota: ci sono molti altri formati che l' `parse` riconosce.

### Nuovo

```
DateTime.new(2014,10,14)
# => #<DateTime: 2014-10-14T00:00:00+00:00 ((2456945j,0s,0n),+0s,2299161j)>
```

Ora attuale:

```
DateTime.now
# => #<DateTime: 2016-08-04T00:43:58-03:00 ((2457605j,13438s,667386397n),-10800s,2299161j)>
```

Si noti che fornisce l'ora corrente nel proprio fuso orario

### Aggiungi / sottrai giorni a DateTime

`DateTime` + `Fixnum` (quantità dei giorni)

```
DateTime.new(2015,12,30,23,0) + 1
# => #<DateTime: 2015-12-31T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` + `Float` (quantità dei giorni)

```
DateTime.new(2015,12,30,23,0) + 2.5
# => #<DateTime: 2016-01-02T11:00:00+00:00 ((2457390j,39600s,0n),+0s,2299161j)>
```

`DateTime` + `Rational` (quantità dei giorni)

```
DateTime.new(2015,12,30,23,0) + Rational(1,2)
# => #<DateTime: 2015-12-31T11:00:00+00:00 ((2457388j,39600s,0n),+0s,2299161j)>
```

`DateTime` - `Fixnum` (quantità giorni)

```
DateTime.new(2015,12,30,23,0) - 1
# => #<DateTime: 2015-12-29T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` - `Float` (numero di giorni)

```
DateTime.new(2015,12,30,23,0) - 2.5
# => #<DateTime: 2015-12-28T11:00:00+00:00 ((2457385j,39600s,0n),+0s,2299161j)>
```

`DateTime` - `Rational` (quantità dei giorni)

```
DateTime.new(2015,12,30,23,0) - Rational(1,2)
# => #<DateTime: 2015-12-30T11:00:00+00:00 ((2457387j,39600s,0n),+0s,2299161j)>
```

Leggi Appuntamento online: https://riptutorial.com/it/ruby/topic/5696/appuntamento

# Capitolo 5: Argomenti della parola chiave

## Osservazioni

**Gli argomenti delle parole chiave sono** stati introdotti in Ruby 2.0 e migliorati in Ruby 2.1 con l'aggiunta degli argomenti delle parole chiave *richieste* .

Un semplice metodo con un argomento di parole chiave è simile al seguente:

```
def say(message: "Hello World")
  puts message
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

Come promemoria, lo stesso metodo senza argomento parola chiave sarebbe stato:

```
def say(message = "Hello World")
  puts message
end

say
# => "Hello World"

say "Today is Monday"
# => "Today is Monday"
```

2.0

È possibile simulare l'argomento delle parole chiave nelle versioni precedenti di Ruby utilizzando un parametro Hash. Questa è ancora una pratica molto comune, specialmente nelle librerie che fornisce compatibilità con le versioni di Ruby precedenti alla 2.0:

```
def say(options = {})
  message = options.fetch(:message, "Hello World")
  puts
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

## Examples

## Utilizzo degli argomenti delle parole chiave

Si definisce un argomento parola chiave in un metodo specificando il nome nella definizione del metodo:

```
def say(message: "Hello World")
  puts message
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

È possibile definire più argomenti di parole chiave, l'ordine di definizione è irrilevante:

```
def say(message: "Hello World", before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => "<p>Hello World</p>"

say message: "Today is Monday"
# => "<p>Today is Monday</p>"

say after: "</p><hr>", message: "Today is Monday"
# => "<p>Today is Monday</p><hr>"
```

Gli argomenti delle parole chiave possono essere combinati con argomenti posizionali:

```
def say(message, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Mescolare argomenti di parole chiave con argomenti posizionali era un approccio molto comune prima di Ruby 2.1, perché non era possibile definire gli argomenti chiave richiesti .

Inoltre, in Ruby <2.0, era molto comune aggiungere un `Hash` alla fine di una definizione di metodo da utilizzare per gli argomenti opzionali. La sintassi è molto simile agli argomenti delle parole chiave, al punto che gli argomenti opzionali tramite `Hash` sono compatibili con gli argomenti delle parole chiave di Ruby 2.

```
def say(message, options = {})
  before = option.fetch(:before, "<p>")
  after  = option.fetch(:after, "</p>")
  puts "#{before}#{message}#{after}"
end

# The method call is syntactically equivalent to the keyword argument one
```

```
say "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Si noti che il tentativo di passare un argomento di parole chiave non definito comporterà un errore:

```
def say(message: "Hello World")
  puts message
end

say foo: "Hello"
# => ArgumentError: unknown keyword: foo
```

## Argomenti della parola chiave richiesti

2.1

**Gli argomenti della parola chiave richiesti sono** stati introdotti in Ruby 2.1, come miglioramento degli argomenti delle parole chiave.

Per definire un argomento di parole chiave come richiesto, dichiara semplicemente l'argomento senza un valore predefinito.

```
def say(message:)
  puts message
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "Hello World"
```

Puoi anche mescolare gli argomenti delle parole chiave richiesti e non richiesti:

```
def say(before: "<p>", message:, after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "<p>Hello World</p>"

say message: "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

## Utilizzo di argomenti di parole chiave arbitrarie con operatore splat

È possibile definire un metodo per accettare un numero arbitrario di argomenti parola chiave utilizzando l'operatore *double splat* ( `**` ):

```
def say(**args)
  puts args
end

say foo: "1", bar: "2"
# {:foo=>"1", :bar=>"2"}
```

Gli argomenti sono catturati in un `Hash` . È possibile manipolare l' `Hash` , ad esempio per estrarre gli argomenti desiderati.

```
def say(**args)
  puts args[:message] || "Message not found"
end

say foo: "1", bar: "2", message: "Hello World"
# Hello World

say foo: "1", bar: "2"
# Message not found
```

L'utilizzo dell'operatore splat con gli argomenti delle parole chiave impedirà la convalida degli argomenti delle parole chiave, il metodo non genererà mai `ArgumentError` in caso di parola chiave sconosciuta.

Per quanto riguarda l'operatore standard di splat, puoi convertire nuovamente un `Hash` in argomenti di parole chiave per un metodo:

```
def say(message: nil, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

args = { message: "Hello World", after: "</p><hr>" }
say(**args)
# <p>Hello World</p><hr>

args = { message: "Hello World", foo: "1" }
say(**args)
# => ArgumentError: unknown keyword: foo
```

Questo è generalmente usato quando devi manipolare gli argomenti in entrata e passarli a un metodo sottostante:

```
def inner(foo:, bar:)
  puts foo, bar
end

def outer(something, foo: nil, bar: nil, baz: nil)
  puts something
  params = {}
  params[:foo] = foo || "Default foo"
  params[:bar] = bar || "Default bar"
  inner(**params)
end

outer "Hello:", foo: "Custom foo"
```

```
# Hello:
# Custom foo
# Default bar
```

Leggi Argomenti della parola chiave online: https://riptutorial.com/it/ruby/topic/5253/argomenti-della-parola-chiave

# Capitolo 6: Array

## Sintassi

- a = [] # usando array letterale
- a = Array.new # equivalente all'utilizzo letterale
- a = Array.new (5) # crea una matrice con 5 elementi con valore di zero.
- a = Array.new (5, 0) # crea una matrice con 5 elementi con valore predefinito di 0.

## Examples

### #carta geografica

`#map` , fornita da Enumerable, crea una matrice richiamando un blocco su ciascun elemento e raccogliendo i risultati:

```
[1, 2, 3].map { |i| i * 3 }
# => [3, 6, 9]

['1', '2', '3', '4', '5'].map { |i| i.to_i }
# => [1, 2, 3, 4, 5]
```

La matrice originale non è stata modificata; viene restituito un nuovo array contenente i valori trasformati nello stesso ordine dei valori di origine. `map!` può essere usato se si desidera modificare l'array originale.

Nel metodo `map` puoi chiamare il metodo o usare proc a tutti gli elementi dell'array.

```
# call to_i method on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&:to_i)
# => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# using proc (lambda) on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&->(i){ i.to_i * 2})
# => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

`map` è sinonimo di `collect` .

### Creazione di una matrice con il costruttore letterale []

Gli array possono essere creati racchiudendo un elenco di elementi tra parentesi quadre ( `[` e `]` ). Gli elementi dell'array in questa notazione sono separati da virgole:

```
array = [1, 2, 3, 4]
```

Le matrici possono contenere qualsiasi tipo di oggetti in qualsiasi combinazione senza restrizioni sul tipo:

```
array = [1, 'b', nil, [3, 4]]
```

## Crea una serie di stringhe

Gli array di stringhe possono essere creati usando la sintassi della stringa percentuale di ruby:

```
array = %w(one two three four)
```

Questo è funzionalmente equivalente alla definizione della matrice come:

```
array = ['one', 'two', 'three', 'four']
```

Invece di `%w()` puoi utilizzare altre coppie di delimitatori corrispondenti: `%w{...}` , `%w[...]` o `%w<...>` .

È anche possibile utilizzare delimitatori arbitrari non alfanumerici, ad esempio: `%w!...!` , `%w#...#` o `%w@...@` .

`%W` può essere utilizzato al posto di `%w` per incorporare l'interpolazione delle stringhe. Considera quanto segue:

```
var = 'hello'

%w(#{var}) # => ["\#{var}"]
%W(#{var}) # => ["hello"]
```

Più parole possono essere interpretate evadendo lo spazio con un \.

```
%w(Colorado California New\ York) # => ["Colorado", "California", "New York"]
```

## Crea una matrice di simboli

2.0

```
array = %i(one two three four)
```

Crea l'array `[:one, :two, :three, :four]` .

Invece di `%i(...)` , puoi usare `%i{...}` o `%i[...]` o `%i!...!`

Inoltre, se si desidera utilizzare l'interpolazione, è possibile farlo con `%I`

2.0

```
a = 'hello'
b = 'goodbye'
array_one = %I(#{a} #{b} world)
array_two = %i(#{a} #{b} world)
```

Crea gli array: `array_one = [:hello, :goodbye, :world] array_two = [:"\#{a}", :"\#{b}", :world]`

`array_one = [:hello, :goodbye, :world]` e `array_two = [:"\#{a}", :"\#{b}", :world]`

## Crea array con Array :: new

Una matrice vuota ( `[]` ) può essere creata con il metodo di classe di `Array::new` , `Array::new` :

```
Array.new
```

Per impostare la lunghezza dell'array, passare un argomento numerico:

```
Array.new 3 #=> [nil, nil, nil]
```

Esistono due modi per popolare una matrice con valori predefiniti:

- Passa un valore immutabile come secondo argomento.
- Passa un blocco che ottiene l'indice corrente e genera valori mutabili.

```
Array.new 3, :x #=> [:x, :x, :x]

Array.new(3) { |i| i.to_s } #=> ["0", "1", "2"]

a = Array.new 3, "X"            # Not recommended.
a[1].replace "C"                # a => ["C", "C", "C"]

b = Array.new(3) { "X" }        # The recommended way.
b[1].replace "C"                # b => ["X", "C", "X"]
```

## Manipolazione di elementi di matrice

Aggiunta di elementi:

```
[1, 2, 3] << 4
# => [1, 2, 3, 4]

[1, 2, 3].push(4)
# => [1, 2, 3, 4]

[1, 2, 3].unshift(4)
# => [4, 1, 2, 3]

[1, 2, 3] << [4, 5]
# => [1, 2, 3, [4, 5]]
```

Rimozione di elementi:

```
array = [1, 2, 3, 4]
array.pop
# => 4
array
# => [1, 2, 3]

array = [1, 2, 3, 4]
array.shift
```

```
# => 1
array
# => [2, 3, 4]

array = [1, 2, 3, 4]
array.delete(1)
# => 1
array
# => [2, 3, 4]

array = [1,2,3,4,5,6]
array.delete_at(2) // delete from index 2
# => 3
array
# => [1,2,4,5,6]


array = [1, 2, 2, 2, 3]
array - [2]
# => [1, 3]    # removed all the 2s
array - [2, 3, 4]
# => [1]       # the 4 did nothing
```

Combinare gli array:

```
[1, 2, 3] + [4, 5, 6]
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3].concat([4, 5, 6])
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6] - [2, 3]
# => [1, 4, 5, 6]

[1, 2, 3] | [2, 3, 4]
# => [1, 2, 3, 4]

[1, 2, 3] & [3, 4]
# => [3]
```

Puoi anche moltiplicare gli array, ad es

```
[1, 2, 3] * 2
# => [1, 2, 3, 1, 2, 3]
```

## Array unione, intersezione e differenza

```
x = [5, 5, 1, 3]
y = [5, 2, 4, 3]
```

Union ( | ) contiene elementi di entrambi gli array, con i duplicati rimossi:

```
x | y
=> [5, 1, 3, 2, 4]
```

Intersezione ( `&` ) contiene elementi presenti sia nel primo che nel secondo array:

```
x & y
=> [5, 3]
```

Difference ( `-` ) contiene elementi presenti nel primo array e non presenti nel secondo array:

```
x - y
=> [1]
```

### Array di filtraggio

Spesso vogliamo operare solo su elementi di un array che soddisfano una condizione specifica:

## Selezionare

Restituisce elementi che corrispondono a una condizione specifica

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 } # => [4, 5, 6]
```

## Rifiutare

Restituisce elementi che non corrispondono a una condizione specifica

```
array = [1, 2, 3, 4, 5, 6]
array.reject { |number| number > 3 } # => [1, 2, 3]
```

Sia `#select` e `#reject` restituiscono un array, quindi possono essere concatenati:

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 }.reject { |number| number < 5 }
  # => [5, 6]
```

### Iniettare, ridurre

Inietta e riduci sono nomi diversi per la stessa cosa. In altre lingue queste funzioni sono spesso chiamate pieghe (come foldl o foldr). Questi metodi sono disponibili su ogni oggetto Enumerable.

Inject prende una funzione a due argomenti e la applica a tutte le coppie di elementi nella matrice.

Per l'array `[1, 2, 3]` possiamo aggiungere tutti questi insieme al valore iniziale di zero specificando un valore iniziale e un blocco in questo modo:

```
[1,2,3].reduce(0) {|a,b| a + b} # => 6
```

Qui passiamo alla funzione un valore iniziale e un blocco che dice di aggiungere tutti i valori

insieme. Il blocco viene prima eseguito con `0` come `a` `1` come `b` , quindi prende il risultato di ciò come il successivo `a` , quindi aggiungiamo `1` al secondo valore `2` . Quindi prendiamo il risultato di questo ( `3` ) e lo aggiungiamo all'elemento finale nella lista (anche `3` ) dandoci il nostro risultato ( `6` ).

Se si omette il primo argomento, verrà impostato `a` ad essere il primo elemento della lista, quindi l'esempio di cui sopra è la stessa:

```
[1,2,3].reduce {|a,b| a + b} # => 6
```

Inoltre, invece di passare un blocco con una funzione, possiamo passare una funzione con nome come simbolo, con un valore iniziale o senza. Con questo, l'esempio precedente potrebbe essere scritto come:

```
[1,2,3].reduce(0, :+) # => 6
```

o omettendo il valore iniziale:

```
[1,2,3].reduce(:+) # => 6
```

## Accesso agli elementi

Puoi accedere agli elementi di un array in base ai loro indici. La numerazione dell'indice di matrice inizia da `0` .

```
%w(a b c)[0] # => 'a'
%w(a b c)[1] # => 'b'
```

È possibile ritagliare un array usando l'intervallo

```
%w(a b c d)[1..2] # => ['b', 'c'] (indices from 1 to 2, including the 2)
%w(a b c d)[1...2] # => ['b'] (indices from 1 to 2, excluding the 2)
```

Questo restituisce un nuovo array, ma non influenza l'originale. Ruby supporta anche l'uso di indici negativi.

```
%w(a b c)[-1] # => 'c'
%w(a b c)[-2] # => 'b'
```

Puoi anche combinare gli indici negativi e quelli positivi

```
%w(a b c d e)[1...-1] # => ['b', 'c', 'd']
```

### Altri metodi utili

Utilizzare `first` per accedere al primo elemento di un array:

```
[1, 2, 3, 4].first # => 1
```

O `first(n)` per accedere ai primi `n` elementi restituiti in un array:

```
[1, 2, 3, 4].first(2) # => [1, 2]
```

Allo stesso modo per `last` e `last(n)` :

```
[1, 2, 3, 4].last    # => 4
[1, 2, 3, 4].last(2) # => [3, 4]
```

Usa `sample` per accedere a un elemento casuale in una matrice:

```
[1, 2, 3, 4].sample  # => 3
[1, 2, 3, 4].sample  # => 1
```

O `sample(n)` :

```
[1, 2, 3, 4].sample(2) # => [2, 1]
[1, 2, 3, 4].sample(2) # => [3, 4]
```

## Array bidimensionale

Usando il `Array::new` costruttore `Array::new` , puoi inizializzare un array con una determinata dimensione e un nuovo array in ciascuno dei suoi slot. Gli array interni possono anche avere una dimensione e un valore iniziale.

Ad esempio, per creare una matrice 3x4 di zeri:

```
array = Array.new(3) { Array.new(4) { 0 } }
```

L'array generato sopra appare come questo se stampato con `p` :

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Puoi leggere o scrivere su elementi come questo:

```
x = array[0][1]
array[2][3] = 2
```

## Array e l'operatore splat (*)

L'operatore `*` può essere utilizzato per decomprimere variabili e array in modo che possano essere passati come singoli argomenti a un metodo.

Questo può essere usato per racchiudere un singolo oggetto in una matrice, se non lo è già:

```
def wrap_in_array(value)
  [*value]
end

wrap_in_array(1)
#> [1]

wrap_in_array([1, 2, 3])
#> [1, 2, 3]

wrap_in_array(nil)
#> []
```

Nell'esempio precedente, il metodo `wrap_in_array` accetta un argomento, `value` .

Se `value` è una `Array` , i suoi elementi vengono decompressi e viene creato un nuovo array contenente tali elementi.

Se `value` è un singolo oggetto, viene creato un nuovo array contenente quel singolo oggetto.

Se il `value` è `nil` , viene restituito un array vuoto.

L'operatore splat è particolarmente utile se usato come argomento nei metodi in alcuni casi. Ad esempio, consente di gestire `nil` , valori singoli e matrici in modo coerente:

```
def list(*values)
  values.each do |value|
    # do something with value
    puts value
  end
end

list(100)
#> 100

list([100, 200])
#> 100
#> 200

list(nil)
# nothing is outputted
```

## Decomposizione

Qualsiasi array può essere rapidamente **decomposto** assegnando i suoi elementi a più variabili. Un semplice esempio:

```
arr = [1, 2, 3]
# ---
a = arr[0]
b = arr[1]
c = arr[2]
# --- or, the same
a, b, c = arr
```

Precedendo una variabile con l'operatore *splat* ( `*` ) si mette una matrice di tutti gli elementi che non sono stati catturati da altre variabili. Se non ne rimane, viene assegnato un array vuoto. È possibile utilizzare solo uno splat in un singolo compito:

```
a, *b = arr       # a = 1; b = [2, 3]
a, *b, c = arr    # a = 1; b = [2]; c = 3
a, b, c, *d = arr # a = 1; b = 2; c = 3; d = []
a, *b, *c = arr   # SyntaxError: unexpected *
```

La decomposizione è *sicura* e non genera mai errori. `nil` sono assegnati dove non ci sono abbastanza elementi, corrispondenti al comportamento dell'operatore `[]` quando si accede a un indice fuori limite:

```
arr[9000] # => nil
a, b, c, d = arr # a = 1; b = 2; c = 3; d = nil
```

La `to_ary` tenta di chiamare implicitamente sull'attuale oggetto assegnato. Implementando questo metodo nel tuo tipo hai la possibilità di scomporlo:

```
class Foo
  def to_ary
    [1, 2]
  end
end
a, b = Foo.new # a = 1; b = 2
```

Se l'oggetto che si sta `respond_to?` non `respond_to? to_ary` , viene considerato come un array a elemento singolo:

```
1.respond_to?(:to_ary) # => false
a, b = 1 # a = 1; b = nil
```

La decomposizione può anche essere **annidata** usando un'espressione di decomposizione cancellata `()` al posto di quello che altrimenti sarebbe un singolo elemento:

```
arr = [1, [2, 3, 4], 5, 6]
a, (b, *c), *d = arr # a = 1; b = 2; c = [3, 4]; d = [5, 6]
#   ^^^^^
```

Questo è effettivamente l'opposto di *Splat* .

In realtà, qualsiasi espressione di decomposizione può essere delimitata da `()` . Ma per il primo livello la scomposizione è facoltativa.

```
a, b = [1, 2]
(a, b) = [1, 2] # the same thing
```

**Edge case:** *un singolo identificatore* non può essere utilizzato come modello destrutturante, sia esso esterno o nidificato:

```
(a) = [1] # SyntaxError
a, (b) = [1, [2]] # SyntaxError
```

Quando si assegna **un array letterale** a un'espressione destrutturante, esterno `[]` può essere omesso:

```
a, b = [1, 2]
a, b =  1, 2  # exactly the same
```

Questo è noto come **assegnazione parallela** , ma utilizza la stessa decomposizione sotto il cofano. Ciò è particolarmente utile per lo scambio di valori di variabili senza l'utilizzo di variabili temporanee aggiuntive:

```
t = a; a = b; b = t # an obvious way
a, b = b, a         # an idiomatic way
(a, b) = [b, a]     # ...and how it works
```

I valori vengono acquisiti quando si costruisce il lato destro del compito, quindi l'utilizzo delle stesse variabili di origine e destinazione è relativamente sicuro.

## Trasforma la matrice multidimensionale in una matrice unidimensionale (appiattita)

```
[1, 2, [[3, 4], [5]], 6].flatten  # => [1, 2, 3, 4, 5, 6]
```

Se si dispone di un array multidimensionale e occorre renderlo un array *semplice* (ad esempio unidimensionale), è possibile utilizzare il metodo **`#flatten`** .

## Ottieni elementi di array unici

Nel caso in cui sia necessario leggere gli elementi di un array ***evitando le ripetizioni,*** si utilizza il metodo **`#uniq`** :

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
```

Invece, se vuoi rimuovere tutti gli elementi duplicati da una matrice, puoi usare **`#uniq!`** metodo:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
```

Mentre l'output è lo stesso, **`#uniq!`** memorizza anche il nuovo array:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
a
```

```
#=> [1, 1, 2, 3, 4, 4, 5]

a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
a
#=> [1, 2, 3, 4, 5]
```

## Ottieni tutte le combinazioni / permutazioni di un array

Il metodo di `permutation` , quando chiamato con un blocco produce una matrice bidimensionale che consiste in tutte le sequenze ordinate di una raccolta di numeri.

Se questo metodo viene chiamato senza un blocco, restituirà un `enumerator` . Per convertire in un array, chiama il metodo `to_a` .

| Esempio | Risultato |
|---|---|
| `[1,2,3].permutation` | `#<Enumerator: [1,2,3]:permutation` |
| `[1,2,3].permutation.to_a` | `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]` |
| `[1,2,3].permutation(2).to_a` | `[[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]]` |
| `[1,2,3].permutation(4).to_a` | `[]` -> Nessuna permutazione di lunghezza 4 |

Il metodo di `combination` , d'altro canto, quando viene chiamato con un blocco produce una matrice bidimensionale costituita da tutte le sequenze di una raccolta di numeri. Diversamente dalla permutazione, l'ordine è disatteso in combinazioni. Ad esempio, `[1,2,3]` è uguale a `[3,2,1]`

| Esempio | Risultato |
|---|---|
| `[1,2,3].combination(1)` | `#<Enumerator: [1,2,3]:combination` |
| `[1,2,3].combination(1).to_a` | `[[1],[2],[3]]` |
| `[1,2,3].combination(3).to_a` | `[[1,2,3]]` |
| `[1,2,3].combination(4).to_a` | `[]` -> Nessuna combinazione di lunghezza 4 |

Chiamare il metodo di combinazione da solo risulterà in un enumeratore. Per ottenere un array, chiama il metodo `to_a` .

I metodi `repeated_combination` e `repeated_permutation` sono simili, tranne che lo stesso elemento può essere ripetuto più volte.

Ad esempio le sequenze `[1,1]` , `[1,3,3,1]` , `[3,3,3]` non sarebbero valide in combinazioni e permutazioni regolari.

| Esempio | # Combo |
|---|---|
| `[1,2,3].combination(3).to_a.length` | 1 |
| `[1,2,3].repeated_combination(3).to_a.length` | 6 |
| `[1,2,3,4,5].combination(5).to_a.length` | 1 |
| `[1,2,3].repeated_combination(5).to_a.length` | 126 |

## Crea una matrice di numeri o lettere consecutive

Questo può essere facilmente ottenuto chiamando `Enumerable#to_a` su un oggetto `Range` :

```
(1..10).to_a    #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`(a..b)` significa che includerà tutti i numeri tra aeb. Per escludere l'ultimo numero, utilizzare `a...b`

```
a_range = 1...5
a_range.to_a       #=> [1, 2, 3, 4]
```

o

```
('a'..'f').to_a    #=> ["a", "b", "c", "d", "e", "f"]
('a'...'f').to_a   #=> ["a", "b", "c", "d", "e"]
```

Una comoda scorciatoia per creare un array è `[*a..b]`

```
[*1..10]           #=> [1,2,3,4,5,6,7,8,9,10]
[*'a'..'f']        #=> ["a", "b", "c", "d", "e", "f"]
```

## Rimuovi tutti gli elementi nil da un array con #compact

Se un array ha uno o più elementi `nil` e questi devono essere rimossi, l' `Array#compact` o `Array#compact!` i metodi possono essere utilizzati, come di seguito.

```
array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

#notice that the method returns a new copy of the array with nil removed,
#without affecting the original

array = [ 1, nil, 'hello', nil, '5', 33]

#If you need the original array modified, you can either reassign it

array = array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]
```

```
#Or you can use the much more elegant 'bang' version of the method

array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]
```

Infine, nota che se `#compact` o `#compact!` sono chiamati su un array senza elementi `nil` , questi restituiranno zero.

```
array = [ 'foo', 4, 'life']

array.compact # => nil

array.compact! # => nil
```

## Crea una matrice di numeri

Il modo normale per creare una serie di numeri:

```
numbers = [1, 2, 3, 4, 5]
```

Gli oggetti intervallo possono essere ampiamente utilizzati per creare una serie di numeri:

```
numbers = Array(1..10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

numbers = (1..10).to_a # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`#step` e `#map` methods ci permettono di imporre condizioni sull'intervallo di numeri:

```
odd_numbers = (1..10).step(2).to_a # => [1, 3, 5, 7, 9]

even_numbers = 2.step(10, 2).to_a # => [2, 4, 6, 8, 10]

squared_numbers = (1..10).map { |number| number * number } # => [1, 4, 9, 16, 25, 36, 49, 64,
81, 100]
```

Tutti i metodi precedenti caricano i numeri con entusiasmo. Se devi caricarli pigramente:

```
number_generator = (1..100).lazy # => #<Enumerator::Lazy: 1..100>

number_generator.first(10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## Trasmetti l'array da qualsiasi oggetto

Per ottenere array da qualsiasi oggetto, usa `Kernel#Array` .

Quanto segue è un esempio:

```
Array('something') #=> ["something"]
```

```
Array([2, 1, 5])   #=> [2, 1, 5]
Array(1)           #=> [1]
Array(2..4)        #=> [2, 3, 4]
Array([])          #=> []
Array(nil)         #=> []
```

Ad esempio, è possibile sostituire il metodo `join_as_string` dal seguente codice

```
def join_as_string(arg)
  if arg.instance_of?(Array)
    arg.join(',')
  elsif arg.instance_of?(Range)
    arg.to_a.join(',')
  else
    arg.to_s
  end
end

join_as_string('something') #=> "something"
join_as_string([2, 1, 5])   #=> "2,1,5"
join_as_string(1)           #=> "1"
join_as_string(2..4)        #=> "2,3,4"
join_as_string([])          #=> ""
join_as_string(nil)         #=> ""
```

al seguente codice

```
def join_as_string(arg)
  Array(arg).join(',')
end
```

Leggi Array online: https://riptutorial.com/it/ruby/topic/253/array

# Capitolo 7: Array multidimensionali

## introduzione

Gli array multidimensionali in Ruby sono solo array i cui elementi sono altri array.

L'unico problema è che, poiché gli array Ruby possono contenere elementi di tipi misti, è necessario essere sicuri che la matrice che si sta manipolando sia effettivamente composta da altri array e non, ad esempio, da matrici e stringhe.

## Examples

### Inizializzazione di un array 2D

Ricapitoliamo come inizializzare un array di interi 1 rubino 1D:

```
my_array = [1, 1, 2, 3, 5, 8, 13]
```

Essendo un array 2D semplicemente un array di array, puoi inizializzarlo in questo modo:

```
my_array = [
  [1, 1, 2, 3, 5, 8, 13],
  [1, 4, 9, 16, 25, 36, 49, 64, 81],
  [2, 3, 5, 7, 11, 13, 17]
]
```

### Inizializzazione di un array 3D

Puoi scendere di un livello più in basso e aggiungere un terzo livello di array. Le regole non cambiano:

```
my_array = [
  [
    [1, 1, 2, 3, 5, 8, 13],
    [1, 4, 9, 16, 25, 36, 49, 64, 81],
    [2, 3, 5, 7, 11, 13, 17]
  ],
  [
    ['a', 'b', 'c', 'd', 'e'],
    ['z', 'y', 'x', 'w', 'v']
  ],
  [
    []
  ]
]
```

### Accesso a un array annidato

Accedere al terzo elemento del primo sottotitolo:

```
my_array[1][2]
```

### Array appiattito

Dato un array multidimensionale:

```
my_array = [[1, 2], ['a', 'b']]
```

l'operazione di appiattimento è di decomporre tutti i figli dell'array nell'array root:

```
my_array.flatten

# [1, 2, 'a', 'b']
```

Leggi Array multidimensionali online: https://riptutorial.com/it/ruby/topic/10608/array-multidimensionali

# Capitolo 8: Blocchi e Procs e Lambdas

## Sintassi

- Proc.new ( *blocco* )
- lambda {| args | codice }
- -> (arg1, arg2) {codice}
- object.to_proc
- {| single_arg | codice }
- do | arg, (chiave, valore) | *codice* fine

## Osservazioni

Fai attenzione alla precedenza degli operatori quando hai una linea con più metodi concatenati, come:

```
str = "abcdefg"
puts str.gsub(/./) do |match|
  rand(2).zero? ? match.upcase : match.downcase
end
```

Invece di stampare qualcosa come `abCDeFg` , come ci si aspetterebbe, stampa qualcosa come `#<Enumerator:0x00000000af42b28>` - questo perché `do ... end` ha una precedenza inferiore rispetto ai metodi, il che significa che `gsub` vede solo l'argomento `/./` e non l'argomento del blocco. Restituisce un enumeratore. Il blocco finisce per passare a `puts` , che lo ignora e mostra solo il risultato di `gsub(/./)` .

Per risolvere questo problema, `gsub` racchiudere la chiamata `gsub` tra parentesi o usare `{ ... }` .

## Examples

### proc

```
def call_the_block(&calling); calling.call; end

its_a = proc do |*args|
  puts "It's a..." unless args.empty?
  "beautiful day"
end

puts its_a       #=> "beautiful day"
puts its_a.call  #=> "beautiful day"
puts its_a[1, 2] #=> "It's a..." "beautiful day"
```

Abbiamo copiato il metodo `call_the_block` dall'ultimo esempio. Qui, puoi vedere che un proc è fatto chiamando il metodo `proc` con un blocco. Puoi anche vedere che i blocchi, come i metodi, hanno ritorni impliciti, il che significa che anche procs (e lambda) fanno. Nella definizione di `its_a` , puoi

vedere che i blocchi possono prendere argomenti splat e quelli normali; sono anche in grado di prendere argomenti predefiniti, ma non riesco a pensare a un modo per farlo. Infine, è possibile vedere che è possibile utilizzare più sintassi per chiamare un metodo - il metodo `call` o `[]` operatore.

## lambda

```
# lambda using the arrow syntax
hello_world = -> { 'Hello World!' }
hello_world[]
# 'Hello World!'

# lambda using the arrow syntax accepting 1 argument
hello_world = ->(name) { "Hello #{name}!" }
hello_world['Sven']
# "Hello Sven!"

the_thing = lambda do |magic, ohai, dere|
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
end

the_thing.call(1, 2, 3)
# magic! 1
# ohai 3
# 2 means hello

the_thing.call(1, 2)
# ArgumentError: wrong number of arguments (2 for 3)

the_thing[1, 2, 3, 4]
# ArgumentError: wrong number of arguments (4 for 3)
```

Puoi anche usare `->` per creare e `.()` Per chiamare lambda

```
the_thing = ->(magic, ohai, dere) {
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
}

the_thing.(1, 2, 3)
# => magic! 1
# => ohai 3
# => 2 means hello
```

Qui puoi vedere che una lambda è quasi la stessa di una proc. Tuttavia, ci sono molti avvertimenti:

- L'arbitrio degli argomenti di lambda sono applicati; passando il numero errato di argomenti a un lambda, si genera un `ArgumentError` . Possono ancora avere parametri predefiniti, parametri splat, ecc.
- `return` dall'interno di un lambda ritorna dal lambda, mentre il `return` da un proc ritorna al di fuori del campo di applicazione:

```
def try_proc
  x = Proc.new {
    return # Return from try_proc
  }
  x.call
  puts "After x.call" # this line is never reached
end

def try_lambda
  y = -> {
    return # return from y
  }
  y.call
  puts "After y.call" # this line is not skipped
end

try_proc # No output
try_lambda # Outputs "After y.call"
```

## Oggetti come argomenti di blocco ai metodi

Mettere una `&` ( `&` commerciale) davanti a un argomento lo passerà come il blocco del metodo. Gli oggetti saranno convertiti in un `Proc` utilizzando il metodo `to_proc` .

```
class Greeter
  def to_proc
    Proc.new do |item|
      puts "Hello, #{item}"
    end
  end
end

greet = Greeter.new

%w(world life).each(&greet)
```

Questo è un modello comune in Ruby e molte classi standard lo forniscono.

Ad esempio, `Symbol` s implementa `to_proc` inviandosi all'argomento:

```
# Example implementation
class Symbol
  def to_proc
    Proc.new do |receiver|
      receiver.send self
    end
  end
end
```

Questo abilita l'utile `&:symbol` idioma di `&:symbol` , comunemente usato con oggetti `Enumerable` :

```
letter_counts = %w(just some words).map(&:length)  # [4, 4, 5]
```

## blocchi

I blocchi sono blocchi di codice racchiusi tra parentesi graffe `{}` (di solito per blocchi a linea singola) o `do..end` (usati per blocchi a più linee).

```
5.times { puts "Hello world" } # recommended style for single line blocks

5.times do
    print "Hello "
    puts "world"
end   # recommended style for multi-line blocks

5.times {
    print "hello "
    puts "world" } # does not throw an error but is not recommended
```

Nota: le parentesi hanno una precedenza più alta di `do..end`

# cedimento

I blocchi possono essere utilizzati all'interno di metodi e funzioni utilizzando la parola `yield` :

```
def block_caller
    puts "some code"
    yield
    puts "other code"
end
block_caller { puts "My own block" } # the block is passed as an argument to the method.
#some code
#My own block
#other code
```

Attenzione però se il `yield` viene chiamato senza blocco, genererà un `LocalJumpError` . A tal fine, ruby fornisce un altro metodo chiamato `block_given?` questo ti permette di controllare se un blocco è stato passato prima di chiamare rendimento

```
def block_caller
  puts "some code"
  if block_given?
    yield
  else
    puts "default"
  end
  puts "other code"
end
block_caller
# some code
# default
# other code
block_caller { puts "not defaulted"}
# some code
# not defaulted
# other code
```

`yield` può offrire argomenti anche al blocco

```
def yield_n(n)
  p = yield n if block_given?
  p || n
end
yield_n(12) {|n| n + 7 }
#=> 19
yield_n(4)
#=> 4
```

Mentre questo è un semplice esempio, la `yield` può essere molto utile per consentire l'accesso diretto a variabili di istanza o valutazioni nel contesto di un altro oggetto. Per esempio:

```
class Application
  def configuration
    @configuration ||= Configuration.new
    block_given? ? yield(@configuration) : @configuration
  end
end
class Configuration; end

app = Application.new
app.configuration do |config|
  puts config.class.name
end
# Configuration
#=> nil
app.configuration
#=> #<Configuration:0x2bf1d30>
```

Come puoi vedere usando `yield` in questo modo, il codice `app.configuration.#method_name` più leggibile rispetto a chiamare continuamente `app.configuration.#method_name` . Invece è possibile eseguire tutte le configurazioni all'interno del blocco mantenendo il codice contenuto.

# variabili

Le variabili per i blocchi sono locali al blocco (simili alle variabili delle funzioni), muoiono quando il blocco viene eseguito.

```
my_variable = 8
3.times do |x|
    my_variable = x
    puts my_variable
end
puts my_variable
#=> 0
# 1
# 2
# 8
```

I blocchi non possono essere salvati, muoiono una volta eseguiti. Per salvare i blocchi devi usare `procs` e `lambdas` .

## Conversione in Proc

Gli oggetti che rispondono a `to_proc` possono essere convertiti in proc con l'operatore `&` (che consentirà anche che vengano passati come blocchi).

La classe Symbol definisce `#to_proc` così tenta di chiamare il metodo corrispondente sull'oggetto che riceve come parametro.

```
p [ 'rabbit', 'grass' ].map( &:upcase ) # => ["RABBIT", "GRASS"]
```

Gli oggetti metodo definiscono anche `#to_proc` .

```
output = method( :p )

[ 'rabbit', 'grass' ].map( &output ) # => "rabbit\ngrass"
```

## Applicazione parziale e Currying

Tecnicamente, Ruby non ha funzioni, ma metodi. Tuttavia, un metodo Ruby si comporta quasi identicamente a funzioni in altre lingue:

```
def double(n)
  n * 2
end
```

Questo metodo / funzione normale prende un parametro `n` , lo raddoppia e restituisce il valore. Ora definiamo una funzione (o metodo) di ordine superiore:

```
def triple(n)
  lambda {3 * n}
end
```

Invece di restituire un numero, `triple` restituisce un metodo. Puoi testarlo usando Interactive Ruby Shell :

```
$ irb --simple-prompt
>> def double(n)
>>   n * 2
>> end
=> :double
>> def triple(n)
>>   lambda {3 * n}
>> end
=> :triple
>> double(2)
=> 4
>> triple(2)
=> #<Proc:0x007fd07f07bdc0@(irb):7 (lambda)>
```

Se vuoi effettivamente ottenere il numero triplicato, devi chiamare (o "ridurre") il lambda:

```
triple_two = triple(2)
triple_two.call # => 6
```

O più concisamente:

```
triple(2).call
```

## Currying e applicazioni parziali

Questo non è utile in termini di definizione di funzionalità di base, ma è utile se si desidera avere metodi / funzioni che non vengono immediatamente chiamati o ridotti. Ad esempio, supponiamo di voler definire metodi che aggiungono un numero per un numero specifico (ad esempio `add_one(2) = 3` ). Se dovessi definire una tonnellata di questi potresti fare:

```
def add_one(n)
  n + 1
end

def add_two(n)
  n + 2
end
```

Tuttavia, potresti anche fare questo:

```
add = -> (a, b) { a + b }
add_one = add.curry.(1)
add_two = add.curry.(2)
```

Usando il calcolo lambda possiamo dire che l' `add` è `(λa.(λb.(a+b)))` . Il currying è un modo di *applicare parzialmente l'* `add` . Quindi `add.curry.(1)` , is `(λa.(λb.(a+b)))(1)` che può essere ridotto a `(λb.(1+b))` . Applicazione parziale significa che abbiamo passato un argomento da `add` ma abbiamo lasciato l'altro argomento da fornire in seguito. L'output è un metodo specializzato.

## Altri esempi utili di curry

Diciamo che abbiamo una formula generale davvero grande, che se specifichiamo determinati argomenti ad essa, possiamo ottenere formule specifiche da esso. Considera questa formula:

```
f(x, y, z) = sin(x\*y)*sin(y\*z)*sin(z\*x)
```

Questa formula è fatta per lavorare in tre dimensioni, ma diciamo che vogliamo solo questa formula per quanto riguarda y e z. Diciamo anche che per ignorare x, vogliamo impostare il suo valore su pi / 2. Facciamo prima la formula generale:

```
f = ->(x, y, z) {Math.sin(x*y) * Math.sin(y*z) * Math.sin(z*x)}
```

Ora, usiamo il curry per ottenere la nostra formula `yz` :

```
f_yz = f.curry.(Math::PI/2)
```

Quindi per chiamare il lambda memorizzato in `f_yz` :

```
f_xy.call(some_value_x, some_value_y)
```

Questo è piuttosto semplice, ma diciamo che vogliamo ottenere la formula per `xz` . Come possiamo impostare `y` su `Math::PI/2` se non è l'ultimo argomento? Bene, è un po 'più complicato:

```
f_xz = -> (x,z) {f.curry.(x, Math::PI/2, z)}
```

In questo caso, dobbiamo fornire segnaposto per il parametro che non stiamo pre-compilando. Per coerenza possiamo scrivere `f_xy` questo modo:

```
f_xy = -> (x,y) {f.curry.(x, y, Math::PI/2)}
```

Ecco come funziona il calcolo lambda per `f_yz` :

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_yz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (π/2) # Reduce =>
f_yz = (λy.(λz.(sin((π/2)*y) * sin(y*z) * sin(z*(π/2)))))
```

Ora diamo un'occhiata a `f_xz`

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_xz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (λt.t) (π/2)  # Reduce =>
f_xz = (λt.(λz.(sin(t*(π/2)) * sin((π/2)*z) * sin(z*t))))
```

Per maggiori informazioni sul calcolo lambda, prova questo .

Leggi Blocchi e Procs e Lambdas online: https://riptutorial.com/it/ruby/topic/474/blocchi-e-procs-e-lambdas

# Capitolo 9: C estensioni

## Examples

### La tua prima estensione

Le estensioni C sono composte da due parti generali:

1. Il codice C stesso.
2. Il file di configurazione dell'estensione.

Per iniziare con la prima estensione, inserisci quanto segue in un file chiamato `extconf.rb` :

```
require 'mkmf'

create_makefile('hello_c')
```

Un paio di cose da sottolineare:

Innanzitutto, il nome `hello_c` è ciò che verrà chiamato l'output dell'estensione compilata. Sarà quello che usi in congiunzione con `require` .

In secondo luogo, il file `extconf.rb` può effettivamente essere chiamato qualsiasi cosa, è solo tradizionalmente ciò che viene usato per costruire gemme che hanno codice nativo, il file che sta per compilare l'estensione è il Makefile generato durante l'esecuzione di `ruby extconf.rb` . Il Makefile predefinito che viene generato compila tutti i file `.c` nella directory corrente.

Metti quanto segue in un file chiamato `hello.c` ed esegui `ruby extconf.rb && make`

```
#include <stdio.h>
#include "ruby.h"

VALUE world(VALUE self) {
  printf("Hello World!\n");
  return Qnil;
}

// The initialization method for this module
void Init_hello_c() {
  VALUE HelloC = rb_define_module("HelloC");
  rb_define_singleton_method(HelloC, "world", world, 0);
}
```

Una ripartizione del codice:

Il nome `Init_hello_c` deve corrispondere al nome definito nel file `extconf.rb` , altrimenti quando si carica dinamicamente l'estensione, Ruby non sarà in grado di trovare il simbolo per eseguire il bootstrap dell'estensione.

La chiamata a `rb_define_module` sta creando un modulo Ruby di nome `HelloC` che andrà a trovare il

nostro spazio dei nomi sotto le funzioni C.

Infine, la chiamata a `rb_define_singleton_method` rende un metodo a livello di modulo legato direttamente al modulo `HelloC` che possiamo richiamare da ruby con `HelloC.world` .

Dopo aver compilato l'estensione con la chiamata per `make` in `make` che possiamo eseguire il codice nella nostra estensione C.

Accendi una console!

```
irb(main):001:0> require './hello_c'
=> true
irb(main):002:0> HelloC.world
Hello World!
=> nil
```

## Lavorare con C Structs

Per poter lavorare con le strutture C come oggetti Ruby, è necessario `Data_Wrap_Struct` con chiamate a `Data_Wrap_Struct` e `Data_Get_Struct` .

`Data_Wrap_Struct` una struttura dati C in un oggetto Ruby. Ci vuole un puntatore alla struttura dei dati, insieme ad alcuni riferimenti alle funzioni di callback e restituisce un VALORE. La macro `Data_Get_Struct` accetta VALUE e restituisce un puntatore alla struttura dati C.

Ecco un semplice esempio:

```
#include <stdio.h>
#include <ruby.h>

typedef struct example_struct {
  char *name;
} example_struct;

void example_struct_free(example_struct * self) {
  if (self->name != NULL) {
    free(self->name);
  }
  ruby_xfree(self);
}

static VALUE rb_example_struct_alloc(VALUE klass) {
  return Data_Wrap_Struct(klass, NULL, example_struct_free,
ruby_xmalloc(sizeof(example_struct)));
}

static VALUE rb_example_struct_init(VALUE self, VALUE name) {
  example_struct* p;

  Check_Type(name, T_STRING);

  Data_Get_Struct(self, example_struct, p);
  p->name = (char *)malloc(RSTRING_LEN(name) + 1);
  memcpy(p->name, StringValuePtr(name), RSTRING_LEN(name) + 1);
```

```
  return self;
}

static VALUE rb_example_struct_name(VALUE self) {
  example_struct* p;
  Data_Get_Struct(self, example_struct, p);

  printf("%s\n", p->name);

  return Qnil;
}

void Init_example()
{
  VALUE mExample = rb_define_module("Example");
  VALUE cStruct = rb_define_class_under(mExample, "Struct", rb_cObject);

  rb_define_alloc_func(cStruct, rb_example_struct_alloc);
  rb_define_method(cStruct, "initialize", rb_example_struct_init, 1);
  rb_define_method(cStruct, "name", rb_example_struct_name, 0);
}
```

E il file `extconf.rb` :

```
require 'mkmf'

create_makefile('example')
```

Dopo aver compilato l'estensione:

```
irb(main):001:0> require './example'
=> true
irb(main):002:0> test_struct = Example::Struct.new("Test Struct")
=> #<Example::Struct:0x007fc741965068>
irb(main):003:0> test_struct.name
Test Struct
=> nil
```

## Scrittura Inline C - RubyInLine

RubyInline è un framework che ti consente di incorporare altre lingue all'interno del tuo codice Ruby. Definisce il metodo # inline del modulo, che restituisce un oggetto costruttore. Si passa al builder una stringa contenente codice scritto in una lingua diversa da Ruby e il builder lo trasforma in qualcosa che è possibile chiamare da Ruby.

Quando viene fornito il codice C o C ++ (le due lingue supportate nell'installazione predefinita di RubyInline), gli oggetti del builder scrivono una piccola estensione sul disco, la compila e la carica. Non si ha a che fare con la compilazione da soli, ma è possibile vedere il codice generato e le estensioni compilate nella sottodirectory .ruby_inline della propria directory home.

**Incorpora il codice C direttamente nel tuo programma Ruby:**

- RubyInline (disponibile come gemma di rubino ) crea automaticamente un'estensione

RubyInline non funzionerà da dentro irb

```
#!/usr/bin/ruby -w
    # copy.rb
    require 'rubygems'
    require 'inline'

    class Copier
    inline do |builder|
      builder.c <<END
    void copy_file(const char *source, const char *dest)
    {
      FILE *source_f = fopen(source, "r");
      if (!source_f)
      {
        rb_raise(rb_eIOError, "Could not open source : '%s'", source);
      }

      FILE *dest_f = fopen(dest, "w+");
      if (!dest_f)
      {
        rb_raise(rb_eIOError, "Could not open destination : '%s'", dest);
      }

      char buffer[1024];

      int nread = fread(buffer, 1, 1024, source_f);
      while (nread > 0)
      {
        fwrite(buffer, 1, nread, dest_f);
        nread = fread(buffer, 1, 1024, source_f);
      }
    }
    END
     end
    end
```

La funzione C `copy_file` esiste ora come metodo di istanza di `Copier` :

```
open('source.txt', 'w') { |f| f << 'Some text.' }
Copier.new.copy_file('source.txt', 'dest.txt')
puts open('dest.txt') { |f| f.read }
```

Leggi C estensioni online: https://riptutorial.com/it/ruby/topic/5009/c-estensioni

# Capitolo 10: Caricamento dei file di origine

## Examples

### Richiede il caricamento dei file solo una volta

Il metodo Kernel # richiede il caricamento dei file solo una volta (alcune chiamate da `require` comporteranno il codice in quel file che viene valutato solo una volta). Cerca il tuo ruby `$LOAD_PATH` per trovare il file richiesto se il parametro non è un percorso assoluto. Le estensioni come `.rb` , `.so` , `.o` o `.dll` sono facoltative. I percorsi relativi verranno risolti nella directory di lavoro corrente del processo.

```
require 'awesome_print'
```

Il kernel # require_relative consente di caricare file relativi al file in cui viene chiamato `require_relative` .

```
# will search in directory myproj relative to current source file.
#
require_relative 'myproj/version'
```

### Caricamento automatico dei file di origine

Il metodo `Kernel#autoload` registra il nome del file da caricare (usando `Kernel::require` ) la prima volta che si accede al modulo (che può essere una stringa o un simbolo).

```
autoload :MyModule, '/usr/local/lib/modules/my_module.rb'
```

Il metodo Kernel # autoload? restituisce il nome del file da caricare se il nome è registrato come `autoload` .

```
autoload? :MyModule  #=> '/usr/local/lib/modules/my_module.rb'
```

### Caricamento di file opzionali

Quando i file non sono disponibili, la famiglia `require` genererà un `LoadError` . Questo è un esempio che illustra il caricamento di moduli opzionali solo se esistono.

```
module TidBits

@@unavailableModules = []

[
      { name: 'CoreExtend', file: 'core_extend/lib/core_extend'  } \
    , { name: 'Fs'        , file: 'fs/lib/fs'                    } \
    , { name: 'Options'   , file: 'options/lib/options'          } \
    , { name: 'Susu'      , file: 'susu/lib/susu'                } \
```

```
].each do |lib|
    begin
        require_relative lib[ :file ]
    rescue LoadError
        @@unavailableModules.push lib
    end
end
end # module TidBits
```

## Caricamento di file ripetutamente

Il metodo Kernel # load valuterà il codice nel file specificato. Il percorso di ricerca sarà costruito come `require` . Valuterà quel codice su ogni chiamata successiva a differenza del `require` . Non c'è `load_relative` .

```
load `somefile`
```

## Caricamento di diversi file

Puoi utilizzare qualsiasi tecnica di ruby per creare dinamicamente un elenco di file da caricare. Illustrazione del globbing per i file che iniziano con il `test` , caricati in ordine alfabetico.

```
Dir[ "#{ __dir__ }**/test*.rb" ) ].sort.each do |source|
    require_relative source
end
```

Leggi Caricamento dei file di origine online: https://riptutorial.com/it/ruby/topic/3166/caricamento-dei-file-di-origine

# Capitolo 11: Casting (conversione del tipo)

## Examples

### Casting to a String

```
123.5.to_s    #=> "123.5"
String(123.5) #=> "123.5"
```

Solitamente, `String()` chiamerà semplicemente `#to_s` .

Metodi `Kernel#sprintf` e `String#%` comportano in modo simile a C:

```
sprintf("%s", 123.5) #=> "123.5"
"%s" % 123.5 #=> "123.5"
"%d" % 123.5 #=> "123"
"%.2f" % 123.5 #=> "123.50"
```

### Trasmissione a un numero intero

```
"123.50".to_i     #=> 123
Integer("123.50") #=> 123
```

Una stringa assumerà il valore di qualsiasi numero intero all'inizio, ma non prenderà numeri interi da nessun'altra parte:

```
"123-foo".to_i # => 123
"foo-123".to_i # => 0
```

Tuttavia, c'è una differenza quando la stringa non è un intero valido:

```
"something".to_i     #=> 0
Integer("something") # ArgumentError: invalid value for Integer(): "something"
```

### Casting to a Float

```
"123.50".to_f   #=> 123.5
Float("123.50") #=> 123.5
```

Tuttavia, c'è una differenza quando la stringa non è un `Float` valido:

```
"something".to_f   #=> 0.0
Float("something") # ArgumentError: invalid value for Float(): "something"
```

### Galleggianti e Interi

```
1/2 #=> 0
```

Poiché dividiamo due interi, il risultato è un numero intero. Per risolvere questo problema, abbiamo bisogno di trasmettere almeno uno di quelli a Float:

```
1.0 / 2       #=> 0.5
1.to_f / 2    #=> 0.5
1 / Float(2) #=> 0.5
```

In alternativa, `fdiv` può essere utilizzato per restituire il risultato in virgola mobile della divisione senza eseguire espressamente il cast dell'operando:

```
1.fdiv 2 # => 0.5
```

Leggi Casting (conversione del tipo) online: https://riptutorial.com/it/ruby/topic/219/casting--conversione-del-tipo-

# Capitolo 12: Cattura le eccezioni con Begin / Rescue

## Examples

### Un blocco di gestione degli errori di base

Facciamo una funzione per dividere due numeri, questo è molto fiducioso sul suo input:

```
def divide(x, y)
  return x/y
end
```

Questo funzionerà bene per molti input:

```
> puts divide(10, 2)
5
```

Ma non tutto

```
> puts divide(10, 0)
ZeroDivisionError: divided by 0

> puts divide(10, 'a')
TypeError: String can't be coerced into Fixnum
```

Possiamo riscrivere la funzione avvolgendo l'operazione di divisione rischiosa in un `begin... end` blocco `begin... end` per verificare gli errori, e utilizzare una clausola di `rescue` per emettere un messaggio e restituire `nil` se c'è un problema.

```
def divide(x, y)
  begin
    return x/y
  rescue
    puts "There was an error"
    return nil
  end
end

> puts divide(10, 0)
There was an error

> puts divide(10, 'a')
There was an error
```

### Salvataggio dell'errore

È possibile salvare l'errore se si desidera utilizzarlo nella clausola di `rescue`

```
def divide(x, y)
  begin
    x/y
  rescue => e
    puts "There was a %s (%s)" % [e.class, e.message]
    puts e.backtrace
  end
end

> divide(10, 0)
There was a ZeroDivisionError (divided by 0)
       from (irb):10:in `/'
       from (irb):10
       from /Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'

> divide(10, 'a')
There was a TypeError (String can't be coerced into Fixnum)
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `eval'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/context.rb:380:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:489:in `block (2 levels) in
eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:623:in `signal_status'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:486:in `block in eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:246:in `block (2 levels)
in each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `loop'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `block in
each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in
`each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:485:in `eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:395:in `block in start'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `start'
/Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'
```

## Controllo di diversi errori

Se vuoi fare cose diverse in base al tipo di errore, usa più clausole di `rescue` , ognuna con un tipo di errore diverso come argomento.

```
def divide(x, y)
  begin
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  end
end

> divide(10, 0)
Don't divide by zero!

> divide(10, 'a')
```

```
Division only works on numbers!
```

Se si desidera salvare l'errore per l'utilizzo nel blocco di `rescue` :

```
rescue ZeroDivisionError => e
```

Utilizzare una clausola di `rescue` senza argomento per rilevare errori di un tipo non specificato in un'altra clausola di `rescue` .

```
def divide(x, y)
  begin
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
end

> divide(nil, 2)
Don't do that (NoMethodError)
```

In questo caso, provare a dividere `nil` per 2 non è un `ZeroDivisionError` o un `TypeError` , quindi viene gestito dalla clausola di `rescue` predefinita, che stampa un messaggio per farci sapere che si trattava di un `NoMethodError` .

## Nuovo tentativo

In una clausola di `rescue` , è possibile utilizzare `retry` di eseguire nuovamente la clausola `begin` , presumibilmente dopo aver modificato la circostanza che ha causato l'errore.

```
def divide(x, y)
  begin
    puts "About to divide..."
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    y = 1
    retry
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
end
```

Se passiamo i parametri che sappiamo causeranno un errore `TypeError` , viene eseguita la

clausola `begin` (contrassegnata qui stampando "About to divide") e l'errore viene rilevato come prima e viene restituito `nil` :

```
> divide(10, 'a')
About to divide...
Division only works on numbers!
 => nil
```

Ma se passiamo i parametri che causano un `ZeroDivisionError` , la clausola `begin` viene eseguita, l'errore viene rilevato, il divisore modificato da 0 a 1, quindi `retry` fa `begin` blocco `ZeroDivisionError` (dall'alto), ora con un diverso `y` . La seconda volta non ci sono errori e la funzione restituisce un valore.

```
> divide(10, 0)
About to divide...     # First time, 10 ÷ 0
Don't divide by zero!
About to divide...     # Second time 10 ÷ 1
=> 10
```

## Verifica se non è stato sollevato alcun errore

È possibile utilizzare una clausola `else` per il codice che verrà eseguito se non viene generato alcun errore.

```
def divide(x, y)
  begin
    z = x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  else
    puts "This code will run if there is no error."
    return z
  end
end
```

La clausola `else` non viene eseguita se c'è un errore che trasferisce il controllo a una delle clausole di `rescue` :

```
> divide(10,0)
Don't divide by zero!
=> nil
```

Ma se non viene segnalato alcun errore, viene eseguita la clausola `else` :

```
> divide(10,2)
This code will run if there is no error.
```

```
=> 5
```

Si noti che la clausola `else` non verrà eseguita *se si torna dalla clausola* `begin`

```
def divide(x, y)
  begin
    z = x/y
    return z                 # Will keep the else clause from running!
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  else
    puts "This code will run if there is no error."
    return z
  end
end

> divide(10,2)
=> 5
```

## Codice che dovrebbe sempre funzionare

Utilizzare una clausola di `ensure` se esiste un codice che si desidera eseguire sempre.

```
def divide(x, y)
  begin
    z = x/y
    return z
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  ensure
    puts "This code ALWAYS runs."
  end
end
```

La clausola di `ensure` verrà eseguita quando si verifica un errore:

```
> divide(10, 0)
Don't divide by zero!   # rescue clause
This code ALWAYS runs.   # ensure clause
=> nil
```

E quando non ci sono errori:

```
> divide(10, 2)
This code ALWAYS runs.   # ensure clause
=> 5
```

La clausola di verifica è utile quando si desidera assicurarsi, ad esempio, che i file siano chiusi.

Si noti che, a differenza della clausola `else` , la clausola di `ensure` *viene eseguita* prima che la clausola `begin` o `rescue` restituisca un valore. Se la clausola di `ensure` ha un `return` che sostituirà il valore di `return` di qualsiasi altra clausola!

Leggi Cattura le eccezioni con Begin / Rescue online:
https://riptutorial.com/it/ruby/topic/7327/cattura-le-eccezioni-con-begin---rescue

# Capitolo 13: Classi

## Sintassi

- nome della classe
- # Qualche codice che descrive il comportamento della classe
- fine

## Osservazioni

I nomi delle classi in Ruby sono Costanti, quindi la prima lettera dovrebbe essere una maiuscola.

```
class Cat # correct
end

class dog # wrong, throws an error
end
```

## Examples

### Creare una classe

Puoi definire una nuova classe usando la parola chiave `class` .

```
class MyClass
end
```

Una volta definiti, è possibile creare una nuova istanza utilizzando il `.new` metodo

```
somevar = MyClass.new
# => #<MyClass:0x007fe2b8aa4a18>
```

### Costruttore

Una classe può avere un solo costruttore, ovvero un metodo chiamato `initialize` . Il metodo viene automaticamente richiamato quando viene creata una nuova istanza della classe.

```
class Customer
  def initialize(name)
     @name = name.capitalize
  end
end

sarah = Customer.new('sarah')
sarah.name #=> 'Sarah'
```

## Classi e variabili di istanza

Esistono diversi tipi di variabili speciali che una classe può utilizzare per condividere più facilmente i dati.

Variabili di istanza, precedute da `@` . Sono utili se si desidera utilizzare la stessa variabile con metodi diversi.

```
class Person
  def initialize(name, age)
    my_age = age # local variable, will be destroyed at end of constructor
    @name = name # instance variable, is only destroyed when the object is
  end

  def some_method
    puts "My name is #{@name}." # we can use @name with no problem
  end

  def another_method
    puts "My age is #{my_age}." # this will not work!
  end
end

mhmd = Person.new("Mark", 23)

mhmd.some_method #=> My name is Mark.
mhmd.another_method #=> throws an error
```

Variabile di classe, preceduta da `@@` . Contengono gli stessi valori su tutte le istanze di una classe.

```
class Person
  @@persons_created = 0 # class variable, available to all objects of this class
  def initialize(name)
    @name = name

    # modification of class variable persists across all objects of this class
    @@persons_created += 1
  end

  def how_many_persons
    puts "persons created so far: #{@@persons_created}"
  end
end

mark = Person.new("Mark")
mark.how_many_persons #=> persons created so far: 1
helen = Person.new("Helen")

mark.how_many_persons #=> persons created so far: 2
helen.how_many_persons #=> persons created so far: 2
# you could either ask mark or helen
```

Variabili globali, precedute da `$` . Questi sono disponibili ovunque per il programma, quindi assicurati di usarli con saggezza.

```
$total_animals = 0
```

```
class Cat
  def initialize
    $total_animals += 1
  end
end

class Dog
  def initialize
    $total_animals += 1
  end
end

bob = Cat.new()
puts $total_animals #=> 1
fred = Dog.new()
puts $total_animals #=> 2
```

## Accesso alle variabili di istanza con getter e setter

Abbiamo tre metodi:

1. **`attr_reader`** : utilizzato per consentire la `read` della variabile al di fuori della classe.
2. **`attr_writer`** : usato per consentire la modifica della variabile al di fuori della classe.
3. **`attr_accessor`** : combina entrambi i metodi.

```
class Cat
  attr_reader :age # you can read the age but you can never change it
  attr_writer :name # you can change name but you are not allowed to read
  attr_accessor :breed # you can both change the breed and read it

  def initialize(name, breed)
    @name = name
    @breed = breed
    @age = 2
  end
  def speak
    puts "I'm #{@name} and I am a #{@breed} cat"
  end
end

my_cat = Cat.new("Banjo", "birman")
# reading values:

my_cat.age  #=> 2
my_cat.breed #=> "birman"
my_cat.name #=> Error

# changing values

my_cat.age = 3 #=> Error
my_cat.breed = "sphynx"
my_cat.name = "Bilbo"

my_cat.speak #=> I'm Bilbo and I am a sphynx cat
```

Si noti che i parametri sono simboli. questo funziona creando un metodo.

```
class Cat
  attr_accessor :breed
end
```

È fondamentalmente lo stesso di:

```
class Cat
  def breed
    @breed
  end
  def breed= value
    @breed = value
  end
end
```

### Livelli di accesso

Ruby ha tre livelli di accesso. Sono `public` , `private` e `protected` .

I metodi che seguono le parole chiave `private` o `protected` sono definiti come tali. I metodi che precedono questi sono metodi implicitamente `public` .

## Metodi pubblici

Un metodo pubblico dovrebbe descrivere il comportamento dell'oggetto che si sta creando. Questi metodi possono essere richiamati dall'esterno dell'oggetto creato.

```
class Cat
  def initialize(name)
    @name = name
  end

  def speak
    puts "I'm #{@name} and I'm 2 years old"
  end

  ...
end

new_cat = Cat.new("garfield")
#=> <Cat:0x2321868 @name="garfield">

new_cat.speak
#=> I'm garfield and I'm 2 years old
```

Questi metodi sono metodi ruby pubblici, descrivono il comportamento per l'inizializzazione di un nuovo gatto e il comportamento del metodo speak.

`public` parola chiave `public` non è necessaria, ma può essere utilizzata per sfuggire a `private` o `protected`

```
def MyClass
  def first_public_method
```

```
  end

  private

  def private_method
  end

  public

  def second_public_method
  end
end
```

## Metodi privati

I metodi privati non sono accessibili dall'esterno dell'oggetto. Sono usati internamente dall'oggetto. Usando di nuovo l'esempio del gatto:

```
class Cat
  def initialize(name)
    @name = name
  end

  def speak
    age = calculate_cat_age # here we call the private method
    puts "I'm #{@name} and I'm #{age} years old"
  end

  private
     def calculate_cat_age
       2 * 3 - 4
     end
end

my_cat = Cat.new("Bilbo")
my_cat.speak #=> I'm Bilbo and I'm 2 years old
my_cat.calculate_cat_age #=> NoMethodError: private method `calculate_cat_age' called for
#<Cat:0x2321868 @name="Bilbo">
```

Come puoi vedere nell'esempio sopra, l'oggetto Cat appena creato ha accesso al metodo `calculate_cat_age` internamente. Assegniamo l' `age` variabile al risultato dell'esecuzione del metodo private `calculate_cat_age` che stampa il nome e l'età del gatto sulla console.

Quando proviamo a chiamare il metodo `calculate_cat_age` dall'esterno dell'oggetto `my_cat` , riceviamo un `NoMethodError` perché è privato. Prendilo?

## Metodi protetti

I metodi protetti sono molto simili ai metodi privati. Non è possibile accedere al di fuori dell'istanza dell'oggetto nello stesso modo in cui i metodi privati non possono essere. Tuttavia, usando il metodo `self` ruby, i metodi protetti possono essere chiamati all'interno del contesto di un oggetto dello stesso tipo.

```
class Cat
  def initialize(name, age)
    @name = name
    @age = age
  end

  def speak
    puts "I'm #{@name} and I'm #{@age} years old"
  end

  # this == method allows us to compare two objects own ages.
  # if both Cat's have the same age they will be considered equal.
  def ==(other)
     self.own_age == other.own_age
  end

  protected
     def own_age
        self.age
     end
end

cat1 = Cat.new("ricky", 2)
=> #<Cat:0x007fe2b8aa4a18 @name="ricky", @age=2>

cat2 = Cat.new("lucy", 4)
=> #<Cat:0x008gfb7aa6v67 @name="lucy", @age=4>

cat3 = Cat.new("felix", 2)
=> #<Cat:0x009frbaa8V76 @name="felix", @age=2>
```

Puoi vedere che abbiamo aggiunto un parametro età alla classe cat e creato tre nuovi oggetti gatto con il nome e l'età. Chiameremo il metodo protetto `own_age` per confrontare l'età dei nostri oggetti gatto.

```
cat1 == cat2
=> false

cat1 == cat3
=> true
```

Guardate questo, siamo stati in grado di recuperare l'età di cat1 usando il metodo protetto `self.own_age` e confrontarlo con l'età di `cat2.own_age` chiamando `cat2.own_age` all'interno di cat1.

### Tipi di metodi di classe

Le classi hanno 3 tipi di metodi: istanza, singleton e metodi di classe.

# Metodi di istanza

Questi sono metodi che possono essere chiamati da `instance` della classe.

```
class Thing
  def somemethod
```

```
    puts "something"
  end
end

foo = Thing.new # create an instance of the class
foo.somemethod # => something
```

# Metodo di classe

Questi sono metodi statici, cioè possono essere richiamati sulla classe e non su un'istanza di quella classe.

```
class Thing
  def Thing.hello(name)
    puts "Hello, #{name}!"
  end
end
```

È equivalente usare `self` al posto del nome della classe. Il seguente codice è equivalente al codice sopra:

```
class Thing
  def self.hello(name)
    puts "Hello, #{name}!"
  end
end
```

Invoca il metodo scrivendo

```
Thing.hello("John Doe")  # prints: "Hello, John Doe!"
```

# Metodi Singleton

Questi sono disponibili solo per istanze specifiche della classe, ma non per tutti.

```
# create an empty class
class Thing
end

# two instances of the class
thing1 = Thing.new
thing2 = Thing.new

# create a singleton method
def thing1.makestuff
  puts "I belong to thing one"
end

thing1.makestuff # => prints: I belong to thing one
thing2.makestuff # NoMethodError: undefined method `makestuff' for #<Thing>
```

Entrambi i metodi `singleton` e `class` sono chiamati `eigenclass` es. Fondamentalmente, ciò che ruby fa è creare una classe anonima che contiene tali metodi in modo che non interferisca con le istanze che vengono create.

Un altro modo per farlo è il costruttore della `class <<` . Per esempio:

```
# a class method (same as the above example)
class Thing
  class << self # the anonymous class
    def hello(name)
      puts "Hello, #{name}!"
    end
  end
end

Thing.hello("sarah") # => Hello, sarah!

# singleton method

class Thing
end

thing1 = Thing.new

class << thing1
  def makestuff
    puts "I belong to thing one"
  end
end

thing1.makestuff # => prints: "I belong to thing one"
```

## Creazione di classi dinamiche

Le classi possono essere create dinamicamente attraverso l'uso di `Class.new` .

```
# create a new class dynamically
MyClass = Class.new

# instantiate an object of type MyClass
my_class = MyClass.new
```

Nell'esempio sopra, una nuova classe viene creata e assegnata alla costante `MyClass` . Questa classe può essere istanziata e utilizzata come qualsiasi altra classe.

Il metodo `Class.new` accetta una `Class` che diventerà la superclasse della classe creata dinamicamente.

```
# dynamically create a class that subclasses another
Staffy = Class.new(Dog)

# instantiate an object of type Staffy
lucky = Staffy.new
lucky.is_a?(Staffy) # true
lucky.is_a?(Dog)    # true
```

Il metodo `Class.new` accetta anche un blocco. Il contesto del blocco è la classe appena creata. Ciò consente di definire metodi.

```
Duck =
  Class.new do
    def quack
      'Quack!!'
    end
  end

# instantiate an object of type Duck
duck = Duck.new
duck.quack # 'Quack!!'
```

## Nuovo, allocare e inizializzare

In molte lingue, vengono create nuove istanze di una classe utilizzando una `new` parola chiave speciale. In Ruby, `new` viene anche usato per creare istanze di una classe, ma non è una parola chiave; invece, è un metodo statico / di classe, non diverso da qualsiasi altro metodo statico / di classe. La definizione è all'incirca questa:

```
class MyClass
   def self.new(*args)
     obj = allocate
     obj.initialize(*args) # oversimplied; initialize is actually private
     obj
   end
end
```

`allocate` esegue la vera 'magia' di creare un'istanza non inizializzata della classe

Si noti inoltre che il valore di ritorno di `initialize` viene scartato e invece viene restituito obj. Questo rende immediatamente chiaro il motivo per cui puoi codificare il tuo metodo di inizializzazione senza preoccuparti di restituire te `self` alla fine.

Il `new` metodo "normale" che tutte le classi ottengono da `Class` funziona come sopra, ma è possibile ridefinirlo come più ti piace o definire alternative che funzionano in modo diverso. Per esempio:

```
class MyClass
  def self.extraNew(*args)
    obj = allocate
    obj.pre_initialize(:foo)
    obj.initialize(*args)
    obj.post_initialize(:bar)
    obj
  end
end
```

Leggi Classi online: https://riptutorial.com/it/ruby/topic/264/classi

# Capitolo 14: Coda

## Sintassi

- q = Queue.new
- q.push oggetto
- q << oggetto # uguale a #push
- q.pop # => oggetto

## Examples

### Lavoratori multipli Lavandino singolo

Vogliamo raccogliere dati creati da più lavoratori.

Per prima cosa creiamo una coda:

```
sink = Queue.new
```

Quindi 16 lavoratori generano tutti un numero casuale e lo spingono nel lavandino:

```
(1..16).to_a.map do
  Thread.new do
    sink << rand(1..100)
  end
end.map(&:join)
```

E per ottenere i dati, converti una coda in una matrice:

```
data = [].tap { |a| a << sink.pop until sink.empty? }
```

### Una fonte più lavoratori

Vogliamo elaborare i dati in parallelo.

Compila la fonte con alcuni dati:

```
source = Queue.new
data = (1..100)
data.each { |e| source << e }
```

Quindi crea alcuni lavoratori per elaborare i dati:

```
(1..16).to_a.map do
  Thread.new do
    until source.empty?
      item = source.pop
```

```
      sleep 0.5
      puts "Processed: #{item}"
    end
  end
end.map(&:join)
```

## One Source - Pipeline of Work - One Sink

Vogliamo elaborare i dati in parallelo e spostarli lungo la linea per essere elaborati da altri lavoratori.

Poiché i lavoratori consumano e producono dati, dobbiamo creare due code:

```
first_input_source = Queue.new
first_output_sink  = Queue.new
100.times { |i| first_input_source << i }
```

La prima ondata di lavoratori legge un elemento da `first_input_source` , elabora l'elemento e scrive i risultati in `first_output_sink` :

```
(1..16).to_a.map do
  Thread.new do
    loop do
      item = first_input_source.pop
      first_output_source << item ** 2
      first_output_source << item ** 3
    end
  end
end
```

La seconda ondata di worker usa `first_output_sink` come sorgente di input e legge, processa quindi scrive su un altro sink di output:

```
second_input_source = first_output_sink
second_output_sink  = Queue.new

(1..32).to_a.map do
  Thread.new do
    loop do
      item = second_input_source.pop
      second_output_sink << item * 2
      second_output_sink << item * 3
    end
  end
end
```

Ora `second_output_sink` è il sink, convertiamolo in array:

```
sleep 5 # workaround in place of synchronization
sink = second_output_sink
[].tap { |a| a << sink.pop until sink.empty? }
```

## Inserimento di dati in una coda: #push

```
q = Queue.new
q << "any object including another queue"
# or
q.push :data
```

- Non vi è alcun limite massimo di acqua, le code possono crescere all'infinito.
- `#push` non blocca mai

## Tirare i dati da una coda - #pop

```
q = Queue.new
q << :data
q.pop #=> :data
```

- `#pop` bloccherà fino a quando non ci saranno dati disponibili.
- `#pop` può essere utilizzato per la sincronizzazione.

## Sincronizzazione: dopo un punto nel tempo

```
syncer = Queue.new

a = Thread.new do
  syncer.pop
  puts "this happens at end"
end

b = Thread.new do
  puts "this happens first"
  STDOUT.flush
  syncer << :ok
end

[a, b].map(&:join)
```

## Convertire una coda in una matrice

```
q = Queue.new
q << 1
q << 2

a = Array.new
a << q.pop until q.empty?
```

O una sola nave :

```
[].tap { |array| array < queue.pop until queue.empty? }
```

## Unione di due code

- Per evitare il blocco all'infinito, la lettura dalle code non dovrebbe avvenire durante l'unione dei thread.

- Per evitare la sincronizzazione o l'attesa infinita di una coda mentre altri hanno dati, la lettura dalle code non dovrebbe avvenire sullo stesso thread.

Iniziamo definendo e compilando due code:

```
q1 = Queue.new
q2 = Queue.new
(1..100).each { |e| q1 << e }
(101..200).each { |e| q2 << e }
```

Dovremmo creare un'altra coda e spingere i dati da altri thread al suo interno:

```
merged = Queue.new

[q1, q2].map do |q|
  Thread.new do
    loop do
      merged << q.pop
    end
  end
end
```

Se sai che puoi consumare completamente entrambe le code (la velocità di consumo è superiore alla produzione, non esaurirai la RAM) c'è un approccio più semplice:

```
merged = Queue.new
merged << q1.pop until q1.empty?
merged << q2.pop until q2.empty?
```

Leggi Coda online: https://riptutorial.com/it/ruby/topic/4666/coda

# Capitolo 15: Commenti

## Examples

### Commenti di riga singole e multiple

I commenti sono annotazioni leggibili dal programmatore che vengono ignorate in fase di runtime. Il loro scopo è rendere il codice sorgente più facile da capire.

**Commenti a riga singola**

Il carattere `#` viene utilizzato per aggiungere commenti a riga singola.

```
#!/usr/bin/ruby -w
# This is a single line comment.
puts "Hello World!"
```

Quando viene eseguito, il programma di cui sopra produrrà `Hello World!`

**Commenti multilinea**

I commenti su più righe possono essere aggiunti utilizzando la sintassi `=begin` e `=end` (anche nota come marcatori del blocco dei commenti) come segue:

```
#!/usr/bin/ruby -w
=begin
This is a multiline comment.
Write as many line as you want.
=end
puts "Hello World!"
```

Quando viene eseguito, il programma di cui sopra produrrà `Hello World!`

Leggi Commenti online: https://riptutorial.com/it/ruby/topic/3464/commenti

# Capitolo 16: costanti

## Sintassi

- MY_CONSTANT_NAME = "il mio valore"

## Osservazioni

Le costanti sono utili in Ruby quando hai valori che non vuoi essere erroneamente modificati in un programma, come le chiavi API.

## Examples

### Definire una costante

```
MY_CONSTANT = "Hello, world" # constant
Constant = 'This is also constant' # constant
my_variable = "Hello, venus" # not constatn
```

Il nome costante inizia con la lettera maiuscola. Tutto ciò che inizia con la lettera maiuscola è considerato come `constant` in Ruby. Quindi anche la `class` e il `module` sono costanti. La migliore pratica è usare tutte le maiuscole per dichiarare costante.

### Modifica una costante

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT = "Hullo, world"
```

Il codice sopra genera un avvertimento, perché dovresti usare le variabili se vuoi cambiare i loro valori. Tuttavia è possibile cambiare una lettera alla volta in una costante senza un avviso, come questo:

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT[1] = "u"
```

Ora, dopo aver cambiato la seconda lettera di `MY_CONSTANT` , diventa `"Hullo, world"` .

### Le costanti non possono essere definite nei metodi

```
def say_hi
  MESSAGE = "Hello"
  puts MESSAGE
end
```

Il codice sopra riportato genera un errore: `SyntaxError: (irb):2: dynamic constant assignment` .

## Definisci e modifica le costanti in una classe

```
class Message
  DEFAULT_MESSAGE = "Hello, world"

  def speak(message = nil)
    if message
      puts message
    else
      puts DEFAULT_MESSAGE
    end
  end
end
```

La costante `DEFAULT_MESSAGE` può essere modificata con il seguente codice:

```
Message::DEFAULT_MESSAGE = "Hullo, world"
```

Leggi costanti online: https://riptutorial.com/it/ruby/topic/4093/costanti

# Capitolo 17: Costanti speciali in Ruby

## Examples

### __FILE__

È il percorso relativo al file dalla directory di esecuzione corrente
Supponiamo di avere questa struttura di directory: /home/stackoverflow/script.rb
script.rb contiene:

```
puts __FILE__
```

Se sei dentro / home / stackoverflow ed esegui lo script come `ruby script.rb` allora `__FILE__` mostrerà `script.rb` Se sei dentro / a casa allora verrà emesso `stackoverflow/script.rb`

Molto utile per ottenere il percorso dello script nelle versioni precedenti alla 2.0 in cui `__dir__` non esiste.

**Nota** `__FILE__` non è uguale a `__dir__`

### __dir__

`__dir__` non è una costante ma una funzione
`__dir__` è uguale a `File.dirname(File.realpath(__FILE__))`

### $ PROGRAM_NAME o $ 0

Contiene il nome dello script in esecuzione.
È lo stesso di `__FILE__` se stai eseguendo quello script.

### $$

Il numero del processo di Ruby che esegue questo script

### $ 1, $ 2, ecc

Contiene il subpattern dalla serie di parentesi corrispondente nell'ultimo modello di successo abbinato, senza contare i pattern abbinati in blocchi nidificati che sono già stati abbandonati, o zero se l'ultima corrispondenza del modello non è riuscita. Queste variabili sono tutte di sola lettura.

### ARGV o $ *

Argomenti della riga di comando forniti per lo script. Le opzioni per l'interprete Ruby sono già state rimosse.

### STDIN

L'input standard. Il valore predefinito per $ stdin

### STDOUT

L'output standard. Il valore predefinito per $ stdout

### STDERR

L'output di errore standard. Il valore predefinito per $ stderr

### $ stderr

L'attuale output di errore standard.

### $ stdout

L'attuale output standard

### $ stdin

L'attuale input standard

### ENV

L'oggetto simile a un hash contiene le variabili di ambiente correnti. L'impostazione di un valore in ENV modifica l'ambiente per i processi figli.

Leggi Costanti speciali in Ruby online: https://riptutorial.com/it/ruby/topic/4037/costanti-speciali-in-ruby

# Capitolo 18: Creazione / gestione gemma

## Examples

### File Gemspec

Ogni gemma ha un file nel formato di `<gem name>.gemspec` che contiene i metadati relativi alla gemma e ai suoi file. Il formato di un gemspec è il seguente:

```
Gem::Specification.new do |s|
  # Details about gem. They are added in the format:
  s.<detail name> = <detail value>
end
```

I campi richiesti da RubyGems sono:

O `author = string` o `authors = array`

Usa `author =` se c'è un solo autore e `authors =` quando ce ne sono diversi. Per `authors=` usa un array che elenca i nomi degli autori.

```
files = array
```

Qui `array` è un elenco di tutti i file nella gemma. Questo può anche essere usato con la funzione `Dir[]` , ad esempio se tutti i file si trovano nella directory `/lib/` , quindi puoi usare `files = Dir["/lib/"]` .

```
name = string
```

Qui stringa è solo il nome della tua gemma. Rubygems consiglia alcune regole da seguire quando si nomina la gemma.

1. Usa caratteri di sottolineatura, NESSUN SPAZI
2. Usa solo lettere minuscole
3. Utilizza hypens per l'estensione gem (ad esempio se il tuo gioiello è denominato `example` per un'estensione, lo chiameresti `example-extension` ) in modo che quando è richiesta l'estensione, può essere richiesto come `require "example/extension"` .

RubyGems aggiunge anche "Se pubblichi una gemma su rubygems.org, può essere rimosso se il nome è discutibile, viola la proprietà intellettuale oi contenuti della gemma soddisfano questi criteri. Puoi segnalare tale gemma sul sito di supporto di RubyGems."

```
platform=
```

Non lo so

```
require_paths=
```

Non lo so

```
summary= string
```

String è un riassunto dello scopo delle gemme e qualsiasi cosa tu voglia condividere sulla gemma.

```
version= string
```

L'attuale numero di versione della gemma.

I campi consigliati sono:

```
email = string
```

Un indirizzo email che verrà associato alla gemma.

```
homepage= string
```

Il sito web in cui vive la gemma.

`license=` o `licenses=`

Non lo so

## Costruire una gemma

Una volta creato il tuo gioiello per pubblicarlo devi seguire alcuni passaggi:

1. Costruisci il tuo gioiello con `gem build <gem name>.gemspec` (il file gemspec deve esistere)
2. Crea un account RubyGems se non ne hai già uno qui
3. Assicurati che non esistano gemme che condividono il nome delle tue gemme
4. Pubblica la tua gemma con `gem publish <gem name>.<gem version number>.gem`

## dipendenze

Per elencare l'albero delle dipendenze:

```
gem dependency
```

Per elencare quali gemme dipendono da una gemma specifica (per esempio bundler)

```
gem dependency bundler --reverse-dependencies
```

Leggi Creazione / gestione gemma online: https://riptutorial.com/it/ruby/topic/4092/creazione---gestione-gemma

# Capitolo 19: Debug

## Examples

### Passando attraverso il codice con Pry e Byebug

Innanzitutto, è necessario installare la gemma di `pry-byebug` . Esegui questo comando:

```
$ gem install pry-byebug
```

Aggiungi questa riga nella parte superiore del tuo file `.rb` :

```
require 'pry-byebug'
```

Quindi inserisci questa linea dovunque desideri un punto di interruzione:

```
binding.pry
```

Un esempio `hello.rb` :

```
require 'pry-byebug'

def hello_world
  puts "Hello"
  binding.pry # break point here
  puts "World"
end
```

Quando esegui il file `hello.rb` , il programma si fermerà su quella linea. È quindi possibile scorrere il codice con il comando `step` . Digita il nome di una variabile per apprenderne il valore. Esci dal debugger con `exit-program` o `!!!` .

Leggi Debug online: https://riptutorial.com/it/ruby/topic/7691/debug

# Capitolo 20: Design Patterns e Idioms in Ruby

## Examples

### Singleton

Ruby Standard Library ha un modulo Singleton che implementa il pattern Singleton. Il primo passo nella creazione di una classe Singleton è richiedere e includere il modulo **`Singleton`** in una classe:

```
require 'singleton'

class Logger
  include Singleton
end
```

Se provi a creare un'istanza di questa classe come faresti normalmente con una classe regolare, viene sollevata un'eccezione `NoMethodError` . Il costruttore viene reso privato per impedire che altre istanze vengano create accidentalmente:

```
Logger.new

#=> NoMethodError: private method `new' called for AppConfig:Class
```

Per accedere all'istanza di questa classe, è necessario utilizzare l' `instance()` :

```
first, second = Logger.instance, Logger.instance
first == second

#=> true
```

#### Esempio di logger

```
require 'singleton'


class Logger
  include Singleton

  def initialize
    @log = File.open("log.txt", "a")
  end

  def log(msg)
    @log.puts(msg)
  end
end
```

Per utilizzare l'oggetto `Logger` :

```
Logger.instance.log('message 2')
```

**Senza Singleton include**

Le suddette implementazioni singleton possono anche essere eseguite senza l'inclusione del modulo Singleton. Questo può essere ottenuto con il seguente:

```
class Logger
  def self.instance
    @instance ||= new
  end
end
```

che è una notazione abbreviata per quanto segue:

```
class Logger
  def self.instance
    @instance = @instance || Logger.new
  end
end
```

Tuttavia, tieni presente che il modulo Singleton è testato e ottimizzato, pertanto rappresenta l'opzione migliore per implementare il tuo singleton con.

## Osservatore

Il modello di osservatore è un modello di progettazione software in cui un oggetto (chiamato `subject` ) mantiene un elenco dei suoi dipendenti (chiamati `observers` ) e li notifica automaticamente di eventuali cambiamenti di stato, di solito chiamando uno dei loro metodi.

Ruby fornisce un semplice meccanismo per implementare il modello di progettazione di Observer. Il modulo `Observable` fornisce la logica per notificare all'abbonato eventuali modifiche nell'oggetto Observable.

Perché funzioni, l'osservabile deve affermare che è cambiato e avvisare gli osservatori.

Gli oggetti che osservano devono implementare un metodo `update()` , che sarà il callback per l'Observer.

Implementiamo una piccola chat, in cui gli utenti possono iscriversi agli utenti e quando uno di loro scrive qualcosa, gli utenti ricevono una notifica.

```
require "observer"

class Moderator
  include Observable

  def initialize(name)
    @name = name
  end

  def write
```

```
    message = "Computer says: No"
    changed
    notify_observers(message)
  end
end

class Warner
  def initialize(moderator, limit)
    @limit = limit
    moderator.add_observer(self)
  end
end

class Subscriber < Warner
  def update(message)
    puts "#{message}"
  end
end

moderator = Moderator.new("Rupert")
Subscriber.new(moderator, 1)
moderator.write
moderator.write
```

Producendo il seguente output:

```
# Computer says: No
# Computer says: No
```

Abbiamo attivato il metodo di `write` due volte nella classe Moderatore, notificando i suoi iscritti, in questo caso solo uno.

Più iscritti aggiungiamo, più le modifiche si propagheranno.

## Decoratore

Il pattern Decorator aggiunge un comportamento agli oggetti senza influenzare altri oggetti della stessa classe. Il pattern decoratore è un'alternativa utile alla creazione di sottoclassi.

Crea un modulo per ogni decoratore. Questo approccio è più flessibile dell'ereditarietà perché puoi combinare le responsabilità in più combinazioni. Inoltre, poiché la trasparenza consente ai decoratori di essere annidati in modo ricorsivo, consente un numero illimitato di responsabilità.

Supponiamo che la classe Pizza abbia un metodo di costo che restituisce 300:

```
class Pizza
  def cost
    300
  end
end
```

Rappresenta la pizza con uno strato aggiuntivo di scoppio di formaggio e il costo aumenta di 50. L'approccio più semplice consiste nel creare una sottoclasse di `PizzaWithCheese` che restituisca 350 nel metodo di costo.

```
class PizzaWithCheese < Pizza
  def cost
    350
  end
end
```

Successivamente, dobbiamo rappresentare una pizza grande che aggiunge 100 al costo di una pizza normale. Possiamo rappresentarlo utilizzando una sottoclasse LargePizza di Pizza.

```
class LargePizza < Pizza
  def cost
    400
  end
end
```

Potremmo anche avere un ExtraLargePizza che aggiunge un ulteriore costo di 15 al nostro LargePizza. Se dovessimo considerare che questi tipi di pizza potevano essere serviti con formaggio, dovremmo aggiungere le sottoclassi LargePizzaWithChese e ExtraLargePizzaWithCheese. Finiremo con un totale di 6 classi.

Per semplificare l'approccio, utilizzare i moduli per aggiungere dinamicamente il comportamento alla classe Pizza:

Modulo + estensione + super decoratore: ->

```
class Pizza
  def cost
    300
  end
end

module CheesePizza
  def cost
    super + 50
  end
end

module LargePizza
  def cost
    super + 100
  end
end

pizza = Pizza.new         #=> cost = 300
pizza.extend(CheesePizza) #=> cost = 350
pizza.extend(LargePizza)  #=> cost = 450
pizza.cost                #=> cost = 450
```

## delega

L'oggetto proxy viene spesso utilizzato per garantire l'accesso protetto a un altro oggetto, che la logica aziendale interna non vogliamo inquinare con i requisiti di sicurezza.

Supponiamo di voler garantire che solo l'utente con autorizzazioni specifiche possa accedere alla

risorsa.

Definizione del proxy: (garantisce che solo gli utenti che effettivamente possono vedere le prenotazioni saranno in grado di prenotare il servizio di prenotazione)

```
class Proxy
  def initialize(current_user, reservation_service)
    @current_user = current_user
    @reservation_service = reservation_service
  end

  def highest_total_price_reservations(date_from, date_to, reservations_count)
    if @current_user.can_see_reservations?
      @reservation_service.highest_total_price_reservations(
        date_from,
        date_to,
        reservations_count
      )
    else
      []
    end
  end
end
```

Modelli e ReservationService:

```
class Reservation
  attr_reader :total_price, :date

  def initialize(date, total_price)
    @date = date
    @total_price = total_price
  end
end

class ReservationService
  def highest_total_price_reservations(date_from, date_to, reservations_count)
    # normally it would be read from database/external service
    reservations = [
      Reservation.new(Date.new(2014, 5, 15), 100),
      Reservation.new(Date.new(2017, 5, 15), 10),
      Reservation.new(Date.new(2017, 1, 15), 50)
    ]

    filtered_reservations = reservations.select do |reservation|
      reservation.date.between?(date_from, date_to)
    end

    filtered_reservations.take(reservations_count)
  end
end

class User
  attr_reader :name

  def initialize(can_see_reservations, name)
    @can_see_reservations = can_see_reservations
    @name = name
  end
```

```
  def can_see_reservations?
    @can_see_reservations
  end
end
```

Servizio al consumo:

```
class StatsService
  def initialize(reservation_service)
    @reservation_service = reservation_service
  end

  def year_top_100_reservations_average_total_price(year)
    reservations = @reservation_service.highest_total_price_reservations(
      Date.new(year, 1, 1),
      Date.new(year, 12, 31),
      100
    )

    if reservations.length > 0
      sum = reservations.reduce(0) do |memo, reservation|
        memo + reservation.total_price
      end

      sum / reservations.length
    else
      0
    end
  end
end
```

Test:

```
def test(user, year)
  reservations_service = Proxy.new(user, ReservationService.new)
  stats_service = StatsService.new(reservations_service)
  average_price = stats_service.year_top_100_reservations_average_total_price(year)
  puts "#{user.name} will see: #{average_price}"
end

test(User.new(true, "John the Admin"), 2017)
test(User.new(false, "Guest"),         2017)
```

---

**BENEFICI**

- **stiamo evitando qualsiasi modifica in `ReservationService` quando vengono modificate le restrizioni di accesso.**
- **non stiamo mescolando i dati relativi al business ( `date_from` , `date_to` , `reservations_count` ) con concetti non collegati al dominio (permessi dell'utente) in servizio.**
- **Anche Consumer ( `StatsService` ) è libero dalla logica relativa alle autorizzazioni**

---

**CAVEATS**

- L'interfaccia proxy è sempre esattamente uguale all'oggetto che nasconde, quindi l'utente che utilizza il servizio fornito dal proxy non era nemmeno a conoscenza della presenza del proxy.

Leggi Design Patterns e Idioms in Ruby online: https://riptutorial.com/it/ruby/topic/2081/design-patterns-e-idioms-in-ruby

# Capitolo 21: destrutturazione

## Examples

### Panoramica

La maggior parte della magia della destrutturazione usa l'operatore splat ( `*` ).

| Esempio | Risultato / *commento* |
|---|---|
| `a, b = [0,1]` | `a=0, b=1` |
| `a, *rest = [0,1,2,3]` | `a=0, rest=[1,2,3]` |
| `a, * = [0,1,2,3]` | `a=0` *Equivalente a* `.first` |
| `*, z = [0,1,2,3]` | `z=3` *Equivalente a* `.last` |

### Argomenti di blocco distruttivi

```
triples = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

triples.each { |(first, second, third)| puts second }
# 2
# 5
# 8

triples.map { |(first, *rest)| rest.join(' ') } # => ["2 3", "5 6", "8 9"]
```

Leggi destrutturazione online: https://riptutorial.com/it/ruby/topic/4739/destrutturazione

# Capitolo 22: eccezioni

## Osservazioni

*Un'eccezione* è un oggetto che rappresenta il verificarsi di una condizione eccezionale. In altre parole, indica che qualcosa è andato storto.

In Ruby, le *eccezioni* sono spesso definite *errori* . Questo perché la classe `Exception` base esiste come elemento di un oggetto di eccezione di livello superiore, ma le eccezioni di esecuzione definite dall'utente sono generalmente `StandardError` o discendenti.

## Examples

### Alzare un'eccezione

Per generare un'eccezione usa `Kernel#raise` passando la classe di eccezione e / o il messaggio:

```
raise StandardError # raises a StandardError.new
raise StandardError, "An error" # raises a StandardError.new("An error")
```

Puoi anche semplicemente passare un messaggio di errore. In questo caso, il messaggio è avvolto in un `RuntimeError` :

```
raise "An error" # raises a RuntimeError.new("An error")
```

Ecco un esempio:

```
def hello(subject)
  raise ArgumentError, "`subject` is missing" if subject.to_s.empty?
  puts "Hello #{subject}"
end

hello # => ArgumentError: `subject` is missing
hello("Simone") # => "Hello Simone"
```

### Creazione di un tipo di eccezione personalizzato

Un'eccezione personalizzata è qualsiasi classe che estende `Exception` o una sottoclasse di `Exception` .

In generale, si dovrebbe sempre estendere l'errore `StandardError` o un discendente. La famiglia `Exception` è in genere per errori di macchina virtuale o di sistema, il loro salvataggio può impedire un'interruzione forzata di funzionare come previsto.

```
# Defines a new custom exception called FileNotFound
class FileNotFound < StandardError
end
```

```
def read_file(path)
  File.exist?(path) || raise(FileNotFound, "File #{path} not found")
  File.read(path)
end

read_file("missing.txt")  #=> raises FileNotFound.new("File `missing.txt` not found")
read_file("valid.txt")    #=> reads and returns the content of the file
```

È comune denominare le eccezioni aggiungendo il suffisso `Error` alla fine:

- `ConnectionError`
- `DontPanicError`

Tuttavia, quando l'errore è autoesplicativo, non è necessario aggiungere il suffisso `Error` perché sarebbe ridondante:

- `FileNotFound` VS `FileNotFoundError`
- `DatabaseExploded` VS `DatabaseExplodedError`

## Gestire un'eccezione

Usa il blocco `begin/rescue` per rilevare (salvare) un'eccezione e gestirla:

```
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
end
```

Una clausola di `rescue` è analoga a un blocco `catch` in un linguaggio di parentesi graffa come C # o Java.

Un semplice `rescue` come questo salva `StandardError` .

Nota: fare attenzione ad evitare l'intercettazione di `Exception` anziché l'errore `StandardError` predefinito. La classe `Exception` include `SystemExit` e `NoMemoryError` e altre eccezioni serie che di solito non si desidera `NoMemoryError` . Considerare sempre la cattura di `StandardError` (il valore predefinito).

È inoltre possibile specificare la classe di eccezioni da salvare:

```
begin
  # an excecution that may fail
rescue CustomError
  # something to execute in case of CustomError
  # or descendant
end
```

Questa clausola di salvataggio non catturerà alcuna eccezione che non sia un'eccezione `CustomError` .

È inoltre possibile memorizzare l'eccezione in una variabile specifica:

```
begin
  # an excecution that may fail
rescue CustomError => error
  # error contains the exception
  puts error.message # provide human-readable details about what went wrong.
  puts error.backtrace.inspect # return an array of strings that represent the call stack
end
```

Se non si è riusciti a gestire un'eccezione, è possibile aumentarla in qualsiasi momento in un blocco di salvataggio.

```
begin
   #here goes your code
rescue => e
    #failed to handle
    raise e
end
```

Se si desidera ripetere la `begin` blocco, chiamare `retry` :

```
begin
   #here goes your code
rescue StandardError => e
   #for some reason you want to retry you code
   retry
end
```

Puoi essere bloccato in un ciclo se rilevi un'eccezione in ogni nuovo tentativo. Per evitare ciò, limita il tuo `retry_count` a un certo numero di tentativi.

```
retry_count = 0
begin
      # an excecution that may fail
rescue
    if retry_count < 5
        retry_count = retry_count + 1
        retry
    else
        #retry limit exceeds, do something else
    end
```

Puoi anche fornire un blocco `else` o un blocco di `ensure` . Un `else` blocco sarà eseguito quando il `begin` blocco completa senza eccezione generata. Un blocco di `ensure` verrà sempre eseguito. Un `ensure` blocco è analogo ad un `finally` blocco in una lingua parentesi graffa come C # o Java.

```
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
else
  # something to execute in case of success
ensure
```

```
  # something to always execute
end
```

Se si è all'interno di un `def` , `module` o blocco di `class` , non è necessario utilizzare l'istruzione begin.

```
def foo
    ...
rescue
    ...
end
```

## Gestire più eccezioni

È possibile gestire più errori nella stessa dichiarazione di `rescue` :

```
begin
  # an execution that may fail
rescue FirstError, SecondError => e
  # do something if a FirstError or SecondError occurs
end
```

Puoi anche aggiungere più dichiarazioni di `rescue` :

```
begin
  # an execution that may fail
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
rescue => e
  # do something if a StandardError occurs
end
```

L'ordine dei blocchi di `rescue` è rilevante: la prima partita è quella eseguita. Pertanto, se si imposta `StandardError` come prima condizione e tutte le eccezioni ereditate da `StandardError` , le altre istruzioni di `rescue` non verranno mai eseguite.

```
begin
  # an execution that may fail
rescue => e
  # this will swallow all the errors
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
end
```

Alcuni blocchi hanno una gestione delle eccezioni implicita come `def` , `class` e `module` . Questi blocchi ti consentono di saltare l'istruzione di `begin` .

```
def foo
    ...
rescue CustomError
```

```
    ...
ensure
    ...
end
```

### Aggiunta di informazioni a eccezioni (personalizzate)

Potrebbe essere utile includere informazioni aggiuntive con un'eccezione, ad esempio per la registrazione o per consentire la gestione condizionale quando viene rilevata l'eccezione:

```
class CustomError < StandardError
  attr_reader :safe_to_retry

  def initialize(safe_to_retry = false, message = 'Something went wrong')
    @safe_to_retry = safe_to_retry
    super(message)
  end
end
```

Sollevare l'eccezione:

```
raise CustomError.new(true)
```

Cattura l'eccezione e l'accesso alle informazioni aggiuntive fornite:

```
begin
  # do stuff
rescue CustomError => e
  retry if e.safe_to_retry
end
```

Leggi eccezioni online: https://riptutorial.com/it/ruby/topic/940/eccezioni

# Capitolo 23: Enumerabile in Ruby

## introduzione

Modulo Enumberable, sono disponibili una serie di metodi per eseguire attraversamenti, ordinare, cercare ecc. Attraverso la collezione (Array, Hash, Set, HashMap).

## Examples

### Modulo numerabile

```
1. For Loop:
CountriesName = ["India", "Canada", "America", "Iraq"]
for country in CountriesName
    puts country
end

2. Each Iterator:
Same set of work can be done with each loop which we did with for loop.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each do |country|
    puts country
end


Each iterator, iterate over every single element of the array.
each ---------- iterator
do ------------ start of the block
|country| ----- argument passed to the block
puts country----block



3. each_with_index Iterator:
each_with_index iterator provides the element for the current iteration and index of the
element in that specific collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_with_index do |country, index|
    puts country + "  " + index.to_s
end


4. each_index Iterator:
Just to know the index at which the element is placed in the collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_index do |index|
    puts index
end


5. map:
"map" acts as an iterator and also used to fetch the transformed copy of the array. To fetch
the new set of the array rather than introducing the change in the same specific array.
Let's deal with for loop first:
```

```
You have an array arr = [1,2,3,4,5]
You need to produce new set of array.
arr = [1,2,3,4,5]
newArr = []
for x in 0..arr.length-1
   newArr[x] = -arr[x]
end


The above mentioned array can be iterated and can produce new set of the array using map
method.
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    -x
      end

puts arr
[1,2,3,4,5]

puts newArr
[-1, -2, -3, -4, -5]


map is returning the modified copy of the current value of the collection. arr has unaltered
value.


Difference between each and map:
1. map returned the modified value of the collection.

Let's see the example:
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    puts x
    -x
      end

puts newArr
[-1, -2, -3, -4, -5]

map method is the iterator and also return the copy of transformed collection.


arr = [1,2,3,4,5]
newArr = arr.each do |x|
    puts x
    -x
      end

puts newArr
[1,2,3,4,5]

each block will throw the array because this is just the iterator.
Each iteration, doesn't actually alter each element in the iteration.


6. map!
map with bang changes the orginal collection and returned the modified collection not the copy
of the modified collection.
arr = [1,2,3,4,5]
arr.map! do |x|
```

```
    puts x
    -x
end
puts  arr
[-1, -2, -3, -4, -5]

7. Combining map and each_with_index
Here each_with_index will iterator over the collection and map will return the modified copy
of the collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
newArray =
CountriesName.each_with_index.map do |value, index|
    puts "Value is #{value} and the index is #{index}"
    "Value is #{value} and the index is #{index}"
end

newArray =
CountriesName.each_with_index.map do |value, index|
    if ((index%2).eql?0)
        puts "Value is #{value} and the index is #{index}"
        "Value is #{value} and the index is #{index}"
    end
end

puts newArray
 ["Value is India and the index is 0", nil, "Value is America and the index is 2", nil]


8. select
MixedArray = [1, "India", 2, "Canada", "America", 4]
MixedArray.select do |value|
    (value.class).eql?Integer
end

select method fetches the result based on satifying certain condition.


9. inject methods
inject method reduces the collection to a certain final value.
Let's say you want to find out the sum of the collection.
With for loop how would it work
arr = [1,2,3,4,5]
sum = 0
for x in 0..arr.length-1
   sum = sum + arr[0]
end
puts sum
15

So above mentioned sum can be reduce by single method
arr = [1,2,3,4,5]
arr.inject(0) do |sum, x|
    puts x
    sum = sum + x
end
inject(0) - passing initial value sum = 0
If used inject with no argument sum = arr[0]
sum - After each iteration, total is equal to the return value at the end of the block.
x  - refers to the current iteration's element

inject method is also an iterator.
```

Riepilogo: il modo migliore per trasformare la raccolta consiste nell'utilizzare il modulo Enumerable per compattare il codice clunky.

Leggi Enumerabile in Ruby online: https://riptutorial.com/it/ruby/topic/10786/enumerabile-in-ruby

# Capitolo 24: enumeratori

## introduzione

Un `Enumerator` è un oggetto che implementa l'iterazione in modo controllato.

Anziché eseguire il ciclo fino a quando non viene soddisfatta una condizione, l'oggetto *enumera i* valori secondo necessità. L'esecuzione del ciclo è sospesa fino a quando il proprietario dell'oggetto non richiede il valore successivo.

Gli enumeratori rendono possibili flussi infiniti di valori.

## Parametri

| Parametro | Dettagli |
| --- | --- |
| `yield` | Risponde alla `yield` , che è alias come `<<` . Cedendo a questo oggetto si implementa l'iterazione. |

## Examples

### Enumeratori personalizzati

Creiamo un `Enumerator` per i numeri di Fibonacci .

```
fibonacci = Enumerator.new do |yielder|
  a = b = 1
  loop do
    yielder << a
    a, b = b, a + b
  end
end
```

Ora possiamo usare qualsiasi metodo `Enumerable` con `fibonacci` :

```
fibonacci.take 10
# => [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

### Metodi esistenti

Se un metodo di iterazione come `each` viene chiamato senza un blocco, è necessario restituire un `Enumerator` .

Questo può essere fatto usando il metodo `enum_for` :

```
def each
  return enum_for :each unless block_given?

  yield :x
  yield :y
  yield :z
end
```

Ciò consente al programmatore di comporre operazioni `Enumerable` :

```
each.drop(2).map(&:upcase).first
# => :Z
```

## riavvolgimento

Utilizzare il `rewind` per riavviare l'enumeratore.

```
N = Enumerator.new do |yielder|
  x = 0
  loop do
    yielder << x
    x += 1
  end
end

N.next
# => 0

N.next
# => 1

N.next
# => 2

N.rewind

N.next
# => 0
```

Leggi enumeratori online: https://riptutorial.com/it/ruby/topic/4985/enumeratori

# Capitolo 25: ERB

## introduzione

ERB è l'acronimo di Embedded Ruby e viene utilizzato per inserire variabili Ruby all'interno di modelli, ad esempio HTML e YAML. ERB è una classe Ruby che accetta il testo e valuta e sostituisce il codice Ruby circondato dal markup ERB.

## Sintassi

- <% number = rand (10)%> questo codice sarà valutato
- <% = numero%> questo codice verrà valutato e inserito nell'output
- <% # commento testo%> questo commento non verrà valutato

## Osservazioni

Convegni:

- ERB come modello: logica di business astratta in un codice helper accompagnato, e mantenere i tuoi modelli ERB puliti e leggibili per le persone senza conoscenza di Ruby.
- Aggiungi file con `.erb` : eg `.js.erb` , `.html.erb` , `.css.erb` , ecc.

## Examples

### Parsing ERB

Questo esempio è il testo filtrato da una sessione `IRB` .

```
=> require 'erb'
=> input = <<-HEREDOC
<ul>
<% (0..10).each do |i| %>
    <%# This is a comment %>
    <li><%= i %> is <%= i.even? ? 'even' : 'odd' %>.</li>
<% end %>
</ul>
HEREDOC

=> parser = ERB.new(input)
=> output = parser.result
=> print output
<ul>


    <li>0 is even.</li>


    <li>1 is odd.</li>
```

```
    <li>2 is even.</li>

    <li>3 is odd.</li>

    <li>4 is even.</li>

    <li>5 is odd.</li>

    <li>6 is even.</li>

    <li>7 is odd.</li>

    <li>8 is even.</li>

    <li>9 is odd.</li>

    <li>10 is even.</li>
</ul>
```

Leggi ERB online: https://riptutorial.com/it/ruby/topic/8145/erb

# Capitolo 26: Eredità

## Sintassi

- classe Sottoclasse <SuperClass

## Examples

### Rifattorizzare le classi esistenti per utilizzare l'ereditarietà

Diciamo che abbiamo due classi, `Cat` and `Dog` .

```
class Cat
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Meow"
  end
end

class Dog
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Woof"
  end
end
```

Il metodo `eat` è esattamente lo stesso in queste due classi. Mentre funziona, è difficile da mantenere. Il problema peggiorerà se non ci sono più animali con lo stesso `eat` metodo. L'ereditarietà può risolvere questo problema.

```
class Animal
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  # No sound method
end

class Cat < Animal
  def sound
    puts "Meow"
  end
end
```

```
class Dog < Animal
  def sound
    puts "Woof"
  end
end
```

Abbiamo creato una nuova classe, `Animal` , e spostato il nostro metodo di `eat` in quella classe. Quindi, abbiamo fatto ereditare `Cat` e `Dog` da questa nuova superclasse comune. Ciò elimina la necessità di ripetere il codice

### Eredità multipla

L'ereditarietà multipla è una caratteristica che consente ad una classe di ereditare da più classi (cioè più di un genitore). Ruby non supporta l'ereditarietà multipla. Supporta solo l'ereditarietà singola (cioè la classe può avere solo un genitore), ma puoi usare la *composizione* per costruire classi più complesse usando i moduli .

### sottoclassi

L'ereditarietà consente alle classi di definire un comportamento specifico basato su una classe esistente.

```
class Animal
  def say_hello
    'Meep!'
  end

  def eat
    'Yumm!'
  end
end

class Dog < Animal
  def say_hello
    'Woof!'
  end
end

spot = Dog.new
spot.say_hello # 'Woof!'
spot.eat       # 'Yumm!'
```

In questo esempio:

- `Dog` eredita da `Animal` , rendendolo una *sottoclasse* .
- `Dog` guadagna sia il `say_hello` che `eat` metodi da `Animal` .
- `Dog` sovrascrive il metodo `say_hello` con diverse funzionalità.

### mixins

Le mixine sono un modo meraviglioso per ottenere qualcosa di simile all'ereditarietà multipla. Ci

consente di ereditare o meglio includere i metodi definiti in un modulo in una classe. Questi metodi possono essere inclusi come metodi di istanza o classe. L'esempio seguente illustra questo design.

```
module SampleModule

  def self.included(base)
    base.extend ClassMethods
  end

  module ClassMethods

    def method_static
      puts "This is a static method"
    end

  end

  def insta_method
    puts "This is an instance method"
  end

end

class SampleClass
  include SampleModule
end

sc = SampleClass.new

sc.insta_method

prints "This is an instance method"

sc.class.method_static

prints "This is a static method"
```

## Cosa viene ereditato?

### I metodi sono ereditati

```
class A
  def boo; p 'boo' end
end

class B < A; end

b = B.new
b.boo # => 'boo'
```

### I metodi di classe sono ereditati

```
class A
  def self.boo; p 'boo' end
end
```

```
class B < A; end

p B.boo # => 'boo'
```

**Le costanti sono ereditate**

```
class A
  WOO = 1
end

class B < A; end

p B::WOO # => 1
```

Ma attenzione, possono essere ignorati:

```
class B
  WOO = WOO + 1
end

p B::WOO # => 2
```

**Le variabili di istanza sono ereditate:**

```
class A
  attr_accessor :ho
  def initialize
    @ho = 'haha'
  end
end

class B < A; end

b = B.new
p b.ho # => 'haha'
```

Attenzione, se si sostituiscono i metodi che inizializzano le variabili di istanza senza chiamare `super` , saranno nulli. Continuando dall'alto:

```
class C < A
  def initialize; end
 end

c = C.new
p c.ho    # => nil
```

**Le variabili di istanza di classe non sono ereditate:**

```
class A
    @foo = 'foo'
    class << self
        attr_accessor :foo
    end
```

```
end

class B < A; end

p B.foo # => nil

# The accessor is inherited, since it is a class method
#
B.foo = 'fob' # possible
```

### Le variabili di classe non sono realmente ereditate

Sono condivisi tra la classe base e tutte le sottoclassi come 1 variabile:

```
class A
    @@foo = 0
    def initialize
        @@foo  += 1
        p @@foo
    end
end

class B < A;end

a = A.new # => 1
b = B.new # => 2
```

Quindi continuando dall'alto:

```
class C < A
  def initialize
    @@foo = -10
    p @@foo
  end
end

a = C.new # => -10
b = B.new # => -9
```

Leggi Eredità online: https://riptutorial.com/it/ruby/topic/625/eredita

# Capitolo 27: Espressioni regolari e operazioni basate su Regex

## Examples

### Gruppi, nominati e non.

Ruby estende la sintassi di gruppo standard `(...)` con un gruppo denominato, `(?<name>...)` . Ciò consente l'estrazione per nome invece di dover contare quanti gruppi hai.

```
name_reg = /h(i|ello), my name is (?<name>.*)/i #i means case insensitive

name_input = "Hi, my name is Zaphod Beeblebrox"

match_data = name_reg.match(name_input) #returns either a MatchData object or nil
match_data = name_input.match(name_reg) #works either way

if match_data.nil? #Always check for nil! Common error.
  puts "No match"
else
  match[0] #=> "Hi, my name is Zaphod Beeblebrox"
  match[1] #=> "i" #the first group, (i|ello)
  match[2] #=> "Zaphod Beeblebrox"
  #Because it was a named group, we can get it by name
  match[:name]  #=> "Zaphod Beeblebrox"
  match["name"] #=> "Zaphod Beeblebrox"
  puts "Hello #{match[:name]}!"
end
```

L'indice della partita viene contato in base all'ordine delle parentesi sinistre (con l'intera espressione regolare il primo gruppo nell'indice 0)

```
reg = /(((a)b)c)(d)/
match = reg.match 'abcd'
match[0] #=> "abcd"
match[1] #=> "abc"
match[2] #=> "ab"
match[3] #=> "a"
match[4] #=> "d"
```

### = ~ operatore

```
if /hay/ =~ 'haystack'
  puts "There is hay in the word haystack"
end
```

*Nota:* l'ordine **è significativo** . Anche se `'haystack' =~ /hay/` è nella maggior parte dei casi equivalente, gli effetti collaterali possono essere diversi:

- Le stringhe catturate dai gruppi di cattura con nome vengono assegnate alle variabili locali solo quando viene chiamato `Regexp#=~` ( `regexp =~ str` );
- Poiché l'operando corretto potrebbe essere un oggetto arbitrario, per `regexp =~ str` verrà chiamato `Regexp#=~` o `String#=~` .

Si noti che questo non restituisce un valore vero / falso, invece restituisce l'indice della corrispondenza se trovato, o zero se non trovato. Perché tutti gli interi in ruby sono veri (compreso 0) e nil è falsi, questo funziona. Se si desidera un valore booleano, utilizzare `#===` come mostrato in un altro esempio .

### quantificatori

Quantificatori consente di specificare il conteggio delle stringhe ripetute.

- Zero o uno:

```
/a?/
```

- Zero o molti:

```
/a*/
```

- Uno o molti:

```
/a+/
```

- Numero esatto:

```
/a{2,4}/ # Two, three or four
/a{2,}/  # Two or more
/a{,4}/  # Less than four (including zero)
```

Di default, i quantificatori sono avidi , il che significa che prendono più personaggi che possono mentre stanno ancora facendo una corrispondenza. Normalmente questo non è evidente:

```
/(?<site>.*) Stack Exchange/ =~ 'Motor Vehicle Maintenance & Repair Stack Exchange'
```

Il `site` gruppo di cattura denominato sarà impostato su "Manutenzione e riparazione veicolo" come previsto. Ma se "Stack Exchange" è una parte facoltativa della stringa (perché potrebbe essere "Stack Overflow"), la soluzione ingenua non funzionerà come previsto:

```
/(?<site>.*)( Stack Exchange)?/
```

Questa versione continuerà a combaciare, ma l'acquisizione nominata includerà 'Stack Exchange' dal momento che `*` mangia avidamente quei personaggi. La soluzione è aggiungere un altro punto interrogativo per rendere il `*` pigro:

```
/(?<site>.*?)( Stack Exchange)?/
```

**Aggiunta `?` a qualsiasi quantificatore lo renderà pigro.**

## Classi di caratteri

Descrive intervalli di simboli

È possibile enumerare i simboli in modo esplicito

```
/[abc]/ # 'a' or 'b' or 'c'
```

O usare intervalli

```
/[a-z]/ # from 'a' to 'z'
```

È possibile combinare intervalli e singoli simboli

```
/[a-cz]/ # 'a' or 'b' or 'c' or 'z'
```

Il trattino principale ( `-` ) è trattato come personaggio

```
/[-a-c]/ # '-' or 'a' or 'b' or 'c'
```

Le classi possono essere negative quando precedono i simboli con `^`

```
/[^a-c]/ # Not 'a', 'b' or 'c'
```

Ci sono alcune scorciatoie per classi molto diffuse e personaggi speciali, oltre alle terminazioni di linea

```
^  # Start of line
$  # End of line
\A # Start of string
\Z # End of string, excluding any new line at the end of string
\z # End of string
.  # Any single character
\s # Any whitespace character
\S # Any non-whitespace character
\d # Any digit
\D # Any non-digit
\w # Any word character (letter, number, underscore)
\W # Any non-word character
\b # Any word boundary
```

`\n` sarà capito semplicemente come nuova linea

Per sfuggire a qualsiasi carattere riservato, come `/` o `[]` e altri usare il backslash (barra a sinistra)

```
\\ # => \
```

```
\[\] # => []
```

### Espressioni regolari nelle dichiarazioni del caso

È possibile verificare se una stringa corrisponde a diverse espressioni regolari utilizzando un'istruzione switch.

## Esempio

```
case "Ruby is #1!"
when /\APython/
    puts "Boooo."
when /\ARuby/
    puts "You are right."
else
    puts "Sorry, I didn't understand that."
end
```

Funziona perché le dichiarazioni del caso vengono verificate per l'uguaglianza usando l'operatore `===` , non l'operatore `==` . Quando una regex è sul lato sinistro di un confronto usando `===` , testerà una stringa per vedere se corrisponde.

### Definire un Regexp

Un Regexp può essere creato in tre modi diversi in Ruby.

- usando le barre: `/ /`
- utilizzando `%r{}`
- utilizzando `Regex.new`

```
#The following forms are equivalent
regexp_slash = /hello/
regexp_bracket = %r{hello}
regexp_new = Regexp.new('hello')

string_to_match = "hello world!"

#All of these will return a truthy value
string_to_match =~ regexp_slash    # => 0
string_to_match =~ regexp_bracket  # => 0
string_to_match =~ regexp_new      # => 0
```

### incontro? - Risultato booleano

Restituisce `true` o `false` , che indica se la regexp è corretta o meno senza aggiornare `$~` e altre variabili correlate. Se è presente il secondo parametro, specifica la posizione nella stringa per iniziare la ricerca.

```
/R.../.match?("Ruby")    #=> true
/R.../.match?("Ruby", 1) #=> false
/P.../.match?("Ruby")    #=> false
```

Rubino 2.4+

## Uso rapido comune

Le espressioni regolari vengono spesso utilizzate nei metodi come parametri per verificare se sono presenti altre stringhe o per cercare e / o sostituire stringhe.

Vedrai spesso quanto segue:

```
string = "My not so long string"
string[/so/] # gives so
string[/present/] # gives nil
string[/present/].nil? # gives true
```

Quindi puoi semplicemente usarlo come controllo se una stringa contiene una sottostringa

```
puts "found" if string[/so/]
```

Più avanzato ma ancora breve e veloce: cerca un gruppo specifico usando il secondo parametro, 2 è il secondo in questo esempio perché la numerazione inizia da 1 e non da 0, un gruppo è ciò che è racchiuso tra parentesi.

```
string[/(n.t).+(l.ng)/, 2] # gives long
```

Anche usato spesso: cerca e sostituisci con `sub` o `gsub` , `\1` dà il primo gruppo trovato, `\2` il secondo:

```
string.gsub(/(n.t).+(l.ng)/, '\1 very \2') # My not very long string
```

L'ultimo risultato è ricordato e può essere utilizzato nelle seguenti righe

```
$2 # gives long
```

Leggi Espressioni regolari e operazioni basate su Regex online:
https://riptutorial.com/it/ruby/topic/1357/espressioni-regolari-e-operazioni-basate-su-regex

# Capitolo 28: Filo

## Examples

### Semantica del thread di base

Un nuovo thread separato dall'esecuzione del thread principale può essere creato utilizzando `Thread.new` .

```
thr = Thread.new {
  sleep 1 # 1 second sleep of sub thread
  puts "Whats the big deal"
}
```

Questo avvierà automaticamente l'esecuzione del nuovo thread.

Per bloccare l'esecuzione del thread principale, fino a quando il nuovo thread si arresta, usa `join` :

```
thr.join #=> ... "Whats the big deal"
```

Si noti che il thread potrebbe aver già finito quando si chiama join, nel qual caso l'esecuzione continuerà normalmente. Se un sottoprocesso non viene mai unito e il thread principale viene completato, il sottoprocesso non eseguirà alcun codice rimanente.

### Accesso alle risorse condivise

Utilizzare un mutex per sincronizzare l'accesso a una variabile a cui si accede da più thread:

```
counter = 0
counter_mutex = Mutex.new

# Start three parallel threads and increment counter
3.times.map do |index|
  Thread.new do
    counter_mutex.synchronize { counter += 1 }
  end
end.each(&:join) # Wait for all threads to finish before killing the process
```

In caso contrario, il valore del `counter` attualmente visibile su un thread potrebbe essere modificato da un altro thread.

Esempio **senza** `Mutex` (vedi ad esempio `Thread 0` , dove `Before` e `After` differiscono di più di `1` ):

```
2.2.0 :224 > counter = 0; 3.times.map { |i| Thread.new { puts "[Thread #{i}] Before:
#{counter}"; counter += 1; puts "[Thread #{i}] After: #{counter}"; } }.each(&:join)
[Thread 2] Before: 0
[Thread 0] Before: 0
[Thread 0] After: 2
[Thread 1] Before: 0
```

```
[Thread 1] After: 3
[Thread 2] After: 1
```

Esempio **con** `Mutex` :

```
2.2.0 :226 > mutex = Mutex.new; counter = 0; 3.times.map { |i| Thread.new { mutex.synchronize
{ puts "[Thread #{i}] Before: #{counter}"; counter += 1; puts "[Thread #{i}] After:
#{counter}"; } } }.each(&:join)
[Thread 2] Before: 0
[Thread 2] After: 1
[Thread 1] Before: 1
[Thread 1] After: 2
[Thread 0] Before: 2
[Thread 0] After: 3
```

## Come uccidere un thread

`Thread.kill` usa `Thread.kill` o `Thread.terminate` :

```
thr = Thread.new { ... }
Thread.kill(thr)
```

## Terminare una discussione

Un thread termina se raggiunge la fine del suo blocco di codice. Il modo migliore per terminare anticipatamente un thread è convincerlo a raggiungere la fine del suo blocco di codice. In questo modo, il thread può eseguire il codice di pulizia prima di morire.

Questo thread esegue un ciclo mentre la variabile di istanza continua è true. Imposta questa variabile su false e il thread morirà di morte naturale:

```
require 'thread'

class CounterThread < Thread
  def initialize
    @count = 0
    @continue = true

    super do
      @count += 1 while @continue
      puts "I counted up to #{@count} before I was cruelly stopped."
    end
  end

  def stop
    @continue = false
  end
end

counter = CounterThread.new
sleep 2
counter.stop
```

Leggi Filo online: https://riptutorial.com/it/ruby/topic/995/filo

# Capitolo 29: Flusso di controllo

## Examples

### se, elsif, else e end

Ruby offre le espressioni `if` e `else` previste per la logica di branching, terminate dalla parola chiave `end` :

```
# Simulate flipping a coin
result = [:heads, :tails].sample

if result == :heads
  puts 'The coin-toss came up "heads"'
else
  puts 'The coin-toss came up "tails"'
end
```

In Ruby, le istruzioni `if` sono espressioni che valutano un valore e il risultato può essere assegnato a una variabile:

```
status = if age < 18
           :minor
         else
           :adult
         end
```

Ruby offre anche operatori ternari in stile C ( vedi qui per i dettagli ) che possono essere espressi come:

```
some_statement ? if_true : if_false
```

Ciò significa che l'esempio precedente che usa if-else può anche essere scritto come

```
status = age < 18 ? :minor : :adult
```

Inoltre, Ruby offre la parola chiave `elsif` che accetta un'espressione per abilitare la logica di ramificazione aggiuntiva:

```
label = if shirt_size == :s
          'small'
        elsif shirt_size == :m
          'medium'
        elsif shirt_size == :l
          'large'
        else
          'unknown size'
        end
```

Se nessuna delle condizioni in un `if` / `elsif` catena sono vere, e non c'è nessun `else` la clausola, quindi l'espressione restituisce a zero. Questo può essere utile all'interno dell'interpolazione delle stringhe, poiché `nil.to_s` è la stringa vuota:

```
"user#{'s' if @users.size != 1}"
```

## Valori di verità e Falsy

In Ruby, ci sono esattamente due valori che sono considerati "falsi" e restituiscono false quando testati come condizione per un'espressione `if` . Loro sono:

- `nil`
- booleano `false`

**Tutti gli** altri valori sono considerati "veritieri", tra cui:

- `0` - zero numerico (intero o altro)
- `""` - Stringhe vuote
- `"\n"` - Stringhe contenenti solo spazi bianchi
- `[]` - Array vuoti
- `{}` - hash vuoti

Prendi, ad esempio, il seguente codice:

```
def check_truthy(var_name, var)
  is_truthy = var ? "truthy" : "falsy"
  puts "#{var_name} is #{is_truthy}"
end

check_truthy("false", false)
check_truthy("nil", nil)
check_truthy("0", 0)
check_truthy("empty string", "")
check_truthy("\\n", "\n")
check_truthy("empty array", [])
check_truthy("empty hash", {})
```

Produrrà:

```
false is falsy
nil is falsy
0 is truthy
empty string is truthy
\n is truthy
empty array is truthy
empty hash is truthy
```

## mentre, fino a

Un ciclo `while` esegue il blocco mentre viene soddisfatta la condizione data:

```
i = 0
while i < 5
  puts "Iteration ##{i}"
  i +=1
end
```

Un ciclo `until` esegue il blocco mentre il condizionale è falso:

```
i = 0
until i == 5
  puts "Iteration ##{i}"
  i +=1
end
```

## In linea se / a meno

Un modello comune consiste nell'utilizzare un inline o trailing, `if` o `unless` :

```
puts "x is less than 5" if x < 5
```

Questo è noto come *modificatore* condizionale ed è un modo pratico per aggiungere codice di protezione semplice e ritorni anticipati:

```
def save_to_file(data, filename)
  raise "no filename given" if filename.empty?
  return false unless data.valid?

  File.write(filename, data)
end
```

Non è possibile aggiungere una clausola `else` a questi modificatori. Inoltre, in genere non è raccomandato l'uso di modificatori condizionali all'interno della logica principale - Per i codici complessi si dovrebbe usare normalmente `if` , `elsif` , `else` .

## salvo che

Una dichiarazione comune è `if !(some condition)` . Ruby offre l'alternativa della dichiarazione a `unless` che.

La struttura è esattamente la stessa di un'istruzione `if` , eccetto che la condizione è negativa. Inoltre, l'istruzione a `unless` che non supporta `elsif` , ma supporta `else` :

```
# Prints not inclusive
unless 'hellow'.include?('all')
  puts 'not inclusive'
end
```

## Caso clinico

Ruby usa la parola chiave `case` per le istruzioni switch.

Come da Ruby Docs :

> Le affermazioni di caso sono costituite da una condizione facoltativa, che è nella posizione di un argomento a `case` e da zero o più clausole `when` . La prima clausola `when` corrisponde alla condizione (o per valutare la verità booleana, se la condizione è nulla) "vince" e la sua stanza di codice viene eseguita. Il valore dell'istruzione case è il valore della clausola `when` successful, o `nil` se non esiste tale clausola.
>
> Una dichiarazione di un caso può terminare con `else` clausola. Ciascuno `when` un'istruzione può avere più valori candidati, separati da virgole.

Esempio:

```
case x
when 1,2,3
  puts "1, 2, or 3"
when 10
  puts "10"
else
  puts "Some other number"
end
```

Versione più breve:

```
case x
when 1,2,3 then puts "1, 2, or 3"
when 10 then puts "10"
else puts "Some other number"
end
```

Il valore della clausola `case` viene confrontato con ciascuna clausola `when` utilizzando il metodo `===` (non `==` ). Pertanto può essere utilizzato con una varietà di diversi tipi di oggetti.

È possibile utilizzare un'istruzione `case` con gli intervalli :

```
case 17
when 13..19
  puts "teenager"
end
```

Una dichiarazione di un `case` può essere utilizzata con un Regexp :

```
case "google"
when /oo/
  puts "word contains oo"
end
```

Una dichiarazione del `case` può essere utilizzata con un Proc o lambda:

```
case 44
when -> (n) { n.even? or n < 0 }
  puts "even or less than zero"
```

```
end
```

È possibile utilizzare un'istruzione `case` con Classi :

```
case x
when Integer
  puts "It's an integer"
when String
  puts "It's a string"
end
```

Implementando il metodo `===` puoi creare le tue classi di corrispondenza:

```
class Empty
  def self.===(object)
    !object or "" == object
  end
end

case ""
when Empty
  puts "name was empty"
else
  puts "name is not empty"
end
```

È possibile utilizzare un'istruzione `case` senza un valore con cui confrontare:

```
case
when ENV['A'] == 'Y'
  puts 'A'
when ENV['B'] == 'Y'
  puts 'B'
else
  puts 'Neither A nor B'
end
```

Un'istruzione `case` ha un valore, quindi puoi usarla come argomento metodo o in un compito:

```
description = case 16
              when 13..19 then "teenager"
              else ""
              end
```

## Controllo del ciclo con interruzione, successivo e ripetizione

Il flusso di esecuzione di un blocco Ruby può essere controllato con le istruzioni `break` , `next` e `redo` .

#### break

L'istruzione `break` uscirà immediatamente dal blocco. Qualsiasi istruzione rimanente nel blocco verrà saltata e l'iterazione terminerà:

```
actions = %w(run jump swim exit macarena)
index = 0

while index < actions.length
  action = actions[index]

  break if action == "exit"

  index += 1
  puts "Currently doing this action: #{action}"
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
```

**next**

La `next` istruzione tornerà immediatamente all'inizio del blocco e procederà con la successiva iterazione. Qualsiasi istruzione rimanente nel blocco verrà saltata:

```
actions = %w(run jump swim rest macarena)
index = 0

while index < actions.length
  action = actions[index]
  index += 1

  next if action == "rest"

  puts "Currently doing this action: #{action}"
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: macarena
```

**redo**

L'istruzione `redo` ritorna immediatamente all'inizio del blocco e riprova la stessa iterazione. Qualsiasi istruzione rimanente nel blocco verrà saltata:

```
actions = %w(run jump swim sleep macarena)
index = 0
repeat_count = 0

while index < actions.length
  action = actions[index]
  puts "Currently doing this action: #{action}"

  if action == "sleep"
    repeat_count += 1
    redo if repeat_count < 3
  end

  index += 1
```

```
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: macarena
```

### `Enumerable` iterativa

Oltre ai cicli, queste istruzioni funzionano con i metodi di iterazione Enumerable, come `each` e `map` :

```
[1, 2, 3].each do |item|
  next if item.even?
  puts "Item: #{item}"
end

# Item: 1
# Item: 3
```

## Blocca i valori dei risultati

In entrambe le istruzioni `break` e `next` , può essere fornito un valore, che verrà utilizzato come valore del risultato del blocco:

```
even_value = for value in [1, 2, 3]
  break value if value.even?
end

puts "The first even value is: #{even_value}"

# The first even value is: 2
```

### buttare, prendere

A differenza di molti altri linguaggi di programmazione, le parole chiave `throw` and `catch` non sono correlate alla gestione delle eccezioni in Ruby.

In Ruby, `throw` e `catch` atti un po 'come etichette in altre lingue. Sono usati per cambiare il flusso di controllo, ma non sono correlati a un concetto di "errore" come sono le eccezioni.

```
catch(:out) do
  catch(:nested) do
    puts "nested"
  end

  puts "before"
  throw :out
  puts "will not be executed"
end
```

```
puts "after"
# prints "nested", "before", "after"
```

## Controllo del flusso con istruzioni logiche

Anche se può sembrare controintuitivo, è possibile utilizzare gli operatori logici per determinare se viene eseguita o meno un'istruzione. Per esempio:

```
File.exist?(filename) or STDERR.puts "#{filename} does not exist!"
```

Questo controllerà se il file esiste e stamperà il messaggio di errore solo se non lo è. L'istruzione `or` è pigro, il che significa che smetterà di essere eseguito una volta che è sicuro se il suo valore sia vero o falso. Non appena viene scoperto che il primo termine è vero, non è necessario verificare il valore dell'altro termine. Ma se il primo termine è falso, deve controllare il secondo termine.

Un uso comune è impostare un valore predefinito:

```
glass = glass or 'full' # Optimist!
```

Questo imposta il valore del `glass` su "pieno" se non è già impostato. Più concisamente, è possibile utilizzare la versione simbolica di `or` :

```
glass ||= 'empty' # Pessimist.
```

È anche possibile eseguire la seconda istruzione solo se la prima è falsa:

```
File.exist?(filename) and puts "#{filename} found!"
```

Di nuovo, `and` è pigro, quindi eseguirà la seconda istruzione solo se necessario per arrivare a un valore.

L'operatore `or` ha una precedenza inferiore rispetto a `and` . Allo stesso modo, `||` ha precedenza più bassa di `&&` . Le forme simbolo hanno una precedenza superiore rispetto alle forme di parole. Questo è utile sapere quando si desidera mescolare questa tecnica con l'assegnazione:

```
a = 1 and b = 2
#=> a==1
#=> b==2
```

```
a = 1 && b = 2; puts a, b
#=> a==2
#=> b==2
```

Si noti che la Guida allo stile di Ruby consiglia :

> E `and` `or` parole chiave sono bannati. La minima leggibilità aggiunta non merita l'alta probabilità di introdurre bug sottili. Per le espressioni booleane, usa sempre `&&` e `||`

anziché. Per il controllo del flusso, utilizzare `if` e a `unless` ; `&&` e `||` sono anche accettabili ma meno chiari.

## inizio, fine

L' `begin` blocco è una struttura di controllo che raggruppa più istruzioni.

```
begin
  a = 7
  b = 6
  a * b
end
```

A `begin` blocco restituirà il valore dell'ultima istruzione nel blocco. Il seguente esempio restituirà `3` .

```
begin
  1
  2
  3
end
```

L' `begin` blocco è utile per l'assegnazione condizionale usando l' `||=` gestore, nella quale possono essere necessarie più istruzioni per restituire un risultato.

```
circumference ||=
  begin
    radius = 7
    tau = Math::PI * 2
    tau * radius
  end
```

Può anche essere combinato con altre strutture di blocco come il `rescue` , `ensure` , `while` , `if` , `unless` , ecc. Per fornire un maggiore controllo del flusso del programma.

`Begin` blocchi `Begin` non sono blocchi di codice, come `{ ... }` o `do ... end` ; non possono essere passati alle funzioni.

## return vs. next: ritorno non locale in un blocco

Considera questo frammento *spezzato* :

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    return 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # => 0
```

Ci si potrebbe aspettare che il `return` restituisca un valore per la matrice di risultati di blocco della

`map` . Quindi il valore di ritorno di `foo` sarebbe `[1, 0, 3, 0]` . Invece, **`return` restituisce un valore dal metodo `foo`** . Si noti che `baz` non viene stampato, il che significa che l'esecuzione non ha mai raggiunto quella linea.

`next` con un valore fa il trucco. Funziona come un `return` livello di blocco.

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    next 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # baz
    # => [1, 0, 3, 0]
```

In assenza di un `return` , il valore restituito dal blocco è il valore della sua ultima espressione.

## Operatore di assegnazione Or-Ugual / Conditional (|| =)

Ruby ha un operatore or-equal che consente di assegnare un valore a una variabile se e solo se tale variabile valuta o `nil` o `false` .

```
||= # this is the operator that achieves this.
```

questo operatore con i doppi tubi che rappresentano o e il segno di uguale che rappresenta l'assegnazione di un valore. Potresti pensare che rappresenti qualcosa del genere:

```
x = x || y
```

questo esempio sopra non è corretto. L'operatore or-equal rappresenta effettivamente questo:

```
x || x = y
```

Se `x` restituisce un valore `nil` o `false` `x` viene assegnato il valore di `y` e lasciato invariato altrimenti.

Ecco un caso d'uso pratico dell'operatore or-equal. Immagina di avere una parte del codice che ci si aspetta che invii un messaggio di posta elettronica a un utente. Cosa fai se per qualsiasi motivo non ci sono e-mail per questo utente. Potresti scrivere qualcosa come questo:

```
if user_email.nil?
   user_email = "error@yourapp.com"
end
```

Usando l'operatore or-equal possiamo tagliare l'intera porzione di codice, fornendo un controllo e una funzionalità chiari e chiari.

```
user_email ||= "error@yourapp.com"
```

Nei casi in cui il `false` è un valore valido, è necessario fare attenzione a non ignorarlo accidentalmente:

```
has_been_run = false
has_been_run ||= true
#=> true

has_been_run = false
has_been_run = true if has_been_run.nil?
#=> false
```

## Operatore ternario

Ruby ha un operatore ternario ( `?:` :), Che restituisce uno dei due valori in base a se una condizione viene valutata come verità:

```
conditional ? value_if_truthy : value_if_falsy

value = true
value ? "true" : "false"
#=> "true"

value = false
value ? "true" : "false"
#=> "false"
```

è lo stesso che scrivere `if a then b else c end` , anche se il ternario è preferito

Esempi:

```
puts (if 1 then 2 else 3 end) # => 2

puts 1 ? 2 : 3                # => 2

x = if 1 then 2 else 3 end
puts x                        # => 2
```

## Operatore Flip-Flop

L'operatore flip flop `..` viene utilizzato tra due condizioni in un'istruzione condizionale:

```
(1..5).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4]
```

La condizione diventa `false` *fino* a *quando* la prima parte diventa `true` . Quindi valuta `true` *finché* la seconda parte diventa `true` . Dopodiché passa nuovamente a `false` .

Questo esempio illustra cosa viene selezionato:

```
[1, 2, 2, 3, 4, 4, 5].select do |e|
```

```
  e if (e == 2) .. (e == 4)
end
# => [2, 2, 3, 4]
```

L'operatore flip-flop funziona solo all'interno di ifs (incluso a `unless` ) e dell'operatore ternario. Altrimenti viene considerato come l'operatore di intervallo.

```
(1..5).select do |e|
  (e == 2) .. (e == 4)
end
# => ArgumentError: bad value for range
```

Può passare da `false` a `true` e viceversa più volte:

```
((1..5).to_a * 2).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4, 2, 3, 4]
```

Leggi Flusso di controllo online: https://riptutorial.com/it/ruby/topic/640/flusso-di-controllo

# Capitolo 30: Gamma

## Examples

### Varia come sequenze

L'uso più importante degli intervalli è esprimere una sequenza

**Sintassi:**

```
(begin..end) => this construct will include end value
(begin...end) => this construct will exclude end value
```

o

```
Range.new(begin,end,exclude_end) => exclude_end is by default false
```

Il valore `end` più importante deve essere maggiore `begin` , altrimenti non restituirà nulla.

**Esempi:**

```
(10..1).to_a            #=> []
(1...3)                 #=> [1, 2]
(-6..-1).to_a           #=> [-6, -5, -4, -3, -2, -1]
('a'..'e').to_a         #=> ["a", "b", "c", "d", "e"]
('a'...'e').to_a        #=> ["a", "b", "c", "d"]
Range.new(1,3).to_a     #=> [1, 2, 3]
Range.new(1,3,true).to_a#=> [1, 2]
```

### Iterare su un intervallo

Puoi facilmente fare qualcosa per ogni elemento in un intervallo.

```
(1..5).each do |i|
    print i
end
# 12345
```

### Intervallo tra le date

```
require 'date'

date1 = Date.parse "01/06/2016"
date2 = Date.parse "05/06/2016"

p "Period #{date1.strftime("%d/%m/%Y")} to #{date2.strftime("%d/%m/%Y")}"

(date1..date2).each do |date|
  p date.strftime("%d/%m/%Y")
```

```
end

# "01/06/2016"
# "02/06/2016"
# "03/06/2016"
# "04/06/2016"
# "05/06/2016"
```

Leggi Gamma online: https://riptutorial.com/it/ruby/topic/3427/gamma

# Capitolo 31: Genera un numero casuale

## introduzione

Come generare un numero casuale in Ruby.

## Osservazioni

Alias di Random :: DEFAULT.rand. Questo utilizza un generatore di numeri pseudo-casuali che si avvicina alla casualità vera

## Examples

### Matrice a 6 facce

```
    # Roll a 6 sided die, rand(6) returns a number from 0 to 5 inclusive
    dice_roll_result = 1 + rand(6)
```

### Genera un numero casuale da un intervallo (incluso)

```
# ruby 1.92
lower_limit = 1
upper_limit = 6
Random.new.rand(lower_limit..upper_limit) # Change your range operator to suit your needs
```

Leggi Genera un numero casuale online: https://riptutorial.com/it/ruby/topic/9626/genera-un-numero-casuale

# Capitolo 32: hash

## introduzione

Un hash è un insieme simile a un dizionario di chiavi uniche e dei loro valori. Chiamati anche array associativi, sono simili agli array, ma laddove una matrice usa numeri interi come indice, un hash consente di utilizzare qualsiasi tipo di oggetto. Puoi recuperare o creare una nuova voce in un hash facendo riferimento alla sua chiave.

## Sintassi

- {first_name: "Noel", second_name: "Edmonds"}
- {: first_name => "Noel",: second_name => "Edmonds"}
- {"Nome" => "Noel", "Secondo nome" => "Edmonds"}
- {first_key => first_value, second_key => second_value}

## Osservazioni

Gli hash in Ruby mappano i valori usando una tabella hash.

Qualsiasi oggetto lavabile può essere usato come chiave. Tuttavia, è molto comune utilizzare un `Symbol` in quanto è generalmente più efficiente in diverse versioni di Ruby, a causa della ridotta allocazione degli oggetti.

```
{ key1: "foo", key2: "baz"  }
```

## Examples

### Creare un hash

Un hash in Ruby è un oggetto che implementa una tabella hash , mappando le chiavi ai valori. Ruby supporta una specifica sintassi letterale per definire gli hash utilizzando `{}` :

```
my_hash = {}  # an empty hash
grades = { 'Mark' => 15, 'Jimmy' => 10, 'Jack' => 10 }
```

Un hash può anche essere creato usando il `new` metodo standard:

```
my_hash = Hash.new  # any empty hash
my_hash = {}        # any empty hash
```

Gli hash possono avere valori di qualsiasi tipo, inclusi tipi complessi come matrici, oggetti e altri

hash:

```
mapping = { 'Mark' => 15, 'Jimmy' => [3,4], 'Nika' => {'a' => 3, 'b' => 5} }
mapping['Mark']   # => 15
mapping['Jimmy']  # => [3, 4]
mapping['Nika']   # => {"a"=>3, "b"=>5}
```

Anche le chiavi possono essere di qualsiasi tipo, incluse quelle complesse:

```
mapping = { 'Mark' => 15, 5 => 10, [1, 2] => 9 }
mapping['Mark']  # => 15
mapping[[1, 2]]  # => 9
```

I simboli sono comunemente usati come chiavi hash e Ruby 1.9 ha introdotto una nuova sintassi specifica per abbreviare questo processo. I seguenti hash sono equivalenti:

```
# Valid on all Ruby versions
grades = { :Mark => 15, :Jimmy => 10, :Jack => 10 }
# Valid in Ruby version 1.9+
grades = { Mark: 15, Jimmy: 10, Jack: 10 }
```

Il seguente hash (valido in tutte le versioni di Ruby) è *diverso* , poiché tutte le chiavi sono stringhe:

```
grades = { "Mark" => 15, "Jimmy" => 10, "Jack" => 10 }
```

Sebbene entrambe le versioni di sintassi possano essere miste, quanto segue è sconsigliato.

```
mapping = { :length => 45, width: 10 }
```

Con Ruby 2.2+, esiste una sintassi alternativa per la creazione di un hash con i tasti simbolo (utile soprattutto se il simbolo contiene spazi):

```
grades = { "Jimmy Choo": 10, :"Jack Sparrow": 10 }
# => { :"Jimmy Choo" => 10, :"Jack Sparrow" => 10}
```

## Accesso ai valori

I singoli valori di un hash vengono letti e scritti usando i metodi `[]` e `[]=` :

```
my_hash = { length: 4, width: 5 }

my_hash[:length] #=> => 4

my_hash[:height] = 9

my_hash #=> {:length => 4, :width => 5, :height => 9 }
```

Per impostazione predefinita, l'accesso a una chiave che non è stata aggiunta all'hash restituisce `nil` , il che significa che è sempre possibile tentare di cercare il valore di una chiave:

```
my_hash = {}

my_hash[:age] # => nil
```

Gli hash possono anche contenere chiavi nelle stringhe. Se si tenta di accedere normalmente, verrà restituito un valore `nil` , ma sarà possibile accedervi tramite le relative chiavi stringa:

```
my_hash = { "name" => "user" }

my_hash[:name]    # => nil
my_hash["name"]   # => user
```

Per le situazioni in cui sono previste o necessarie le chiavi, gli hash hanno un metodo di `fetch` che genera un'eccezione quando si accede a una chiave che non esiste:

```
my_hash = {}

my_hash.fetch(:age) #=> KeyError: key not found: :age
```

`fetch` accetta un valore predefinito come secondo argomento, che viene restituito se la chiave non è stata precedentemente impostata:

```
my_hash =  {}
my_hash.fetch(:age, 45) #=> => 45
```

`fetch` può anche accettare un blocco che viene restituito se la chiave non è stata precedentemente impostata:

```
my_hash = {}
my_hash.fetch(:age) { 21 } #=> 21

my_hash.fetch(:age) do |k|
  puts "Could not find #{k}"
end

#=> Could not find age
```

Gli hash supportano anche un metodo `store` come alias per `[]=` :

```
my_hash = {}

my_hash.store(:age, 45)

my_hash #=> { :age => 45 }
```

Puoi anche ottenere tutti i valori di un hash utilizzando il metodo dei `values` :

```
my_hash = { length: 4, width: 5 }

my_hash.values #=> [4, 5]
```

**Nota: questo è solo per Ruby 2.3+** `#dig` è utile per gli `Hash` nidificati. Estrae il valore nidificato specificato dalla sequenza di oggetti idx chiamando dig a ogni passaggio, restituendo nil se qualsiasi passaggio intermedio è nullo.

```
h = { foo: {bar: {baz: 1}}}

h.dig(:foo, :bar, :baz)   # => 1
h.dig(:foo, :zot, :xyz)   # => nil

g = { foo: [10, 11, 12] }
g.dig(:foo, 1)            # => 11
```

## Impostazione dei valori predefiniti

Per impostazione predefinita, il tentativo di cercare il valore per una chiave che non esiste restituirà `nil` . È possibile specificare facoltativamente un altro valore da restituire (o un'azione da eseguire) quando si accede all'hash con una chiave inesistente. Anche se questo è indicato come "il valore predefinito", non è necessario che sia un singolo valore; potrebbe, ad esempio, essere un valore calcolato come la lunghezza della chiave.

Il valore predefinito di un hash può essere passato al suo costruttore:

```
h = Hash.new(0)

h[:hi] = 1
puts h[:hi]  # => 1
puts h[:bye] # => 0 returns default value instead of nil
```

Un valore predefinito può anche essere specificato su un hash già creato:

```
my_hash = { human: 2, animal: 1 }
my_hash.default = 0
my_hash[:plant] # => 0
```

È importante notare che il **valore predefinito non viene copiato** ogni volta che si accede a una nuova chiave, il che può portare a risultati sorprendenti quando il valore predefinito è un tipo di riferimento:

```
# Use an empty array as the default value
authors = Hash.new([])

# Append a book title
authors[:homer] << 'The Odyssey'

# All new keys map to a reference to the same array:
authors[:plato] # => ['The Odyssey']
```

Per ovviare a questo problema, il costruttore Hash accetta un blocco che viene eseguito ogni volta che si accede a una nuova chiave e il valore restituito viene utilizzato come predefinito:

```
authors = Hash.new { [] }
```

```
# Note that we're using += instead of <<, see below
authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"]}
```

Nota che sopra abbiamo dovuto usare + = invece di << perché il valore predefinito non è assegnato automaticamente all'hash; usando << sarebbe stato aggiunto all'array, ma gli autori [: homer] sarebbero rimasti indefiniti:

```
authors[:homer] << 'The Odyssey' # ['The Odyssey']
authors[:homer] # => []
authors # => {}
```

Per poter assegnare valori predefiniti all'accesso, nonché per calcolare valori predefiniti più sofisticati, il blocco predefinito viene passato sia all'hash che alla chiave:

```
authors = Hash.new { |hash, key| hash[key] = [] }

authors[:homer] << 'The Odyssey'
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"], :plato=>[]}
```

Puoi anche utilizzare un blocco predefinito per eseguire un'azione e / o restituire un valore dipendente dalla chiave (o da altri dati):

```
chars = Hash.new { |hash,key| key.length }

chars[:test] # => 4
```

Puoi persino creare hash più complessi:

```
page_views = Hash.new { |hash, key| hash[key] = { count: 0, url: key } }
page_views["http://example.com"][:count] += 1
page_views # => {"http://example.com"=>{:count=>1, :url=>"http://example.com"}}
```

Per impostare l'impostazione predefinita su Proc su un hash *già esistente* , usa `default_proc=` :

```
authors = {}
authors.default_proc = proc { [] }

authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # {:homer=>["The Odyssey"]}
```

## Creazione automatica di un Deep Hash

Hash ha un valore predefinito per le chiavi che vengono richieste ma non esistono (nil):

```
a = {}
p a[ :b ] # => nil
```

Quando si crea un nuovo hash, è possibile specificare il valore predefinito:

```
b = Hash.new 'puppy'
p b[ :b ]            # => 'puppy'
```

Anche Hash.new accetta un blocco, che consente di creare automaticamente hash annidati, come il comportamento di autovivificazione di Perl o `mkdir -p` :

```
# h is the hash you're creating, and k the key.
#
hash = Hash.new { |h, k| h[k] = Hash.new &h.default_proc }
hash[ :a ][ :b ][ :c ] = 3

p hash # => { a: { b: { c: 3 } } }
```

## Modifica di chiavi e valori

Puoi creare un nuovo hash con le chiavi o i valori modificati, infatti puoi anche aggiungere o eliminare i tasti, usando inject (AKA, reduce ). Ad esempio per produrre un hash con chiavi stringificate e valori maiuscoli:

```
fruit = { name: 'apple', color: 'green', shape: 'round' }
# => {:name=>"apple", :color=>"green", :shape=>"round"}

new_fruit = fruit.inject({}) { |memo, (k,v)| memo[k.to_s] = v.upcase; memo }

# => new_fruit is {"name"=>"APPLE", "color"=>"GREEN", "shape"=>"ROUND"}
```

Hash è un enumerabile, in sostanza una raccolta di coppie chiave / valore. Pertanto ha metodi come `each` , `map` e `inject` .

Per ogni coppia chiave / valore nell'hash viene valutato il blocco dato, il valore del memo alla prima esecuzione è il valore seme passato per `inject` , nel nostro caso un hash vuoto, `{}` . Il valore del `memo` per le valutazioni successive è il valore restituito della valutazione dei blocchi precedenti, questo è il motivo per cui modifichiamo il `memo` impostando una chiave con un valore e quindi restituendo il `memo` alla fine. Il valore di ritorno della valutazione dei blocchi finali è il valore di ritorno `inject` , nel nostro caso `memo` .

Per evitare di dover fornire il valore finale, puoi utilizzare invece each_with_object :

```
new_fruit = fruit.each_with_object({}) { |(k,v), memo| memo[k.to_s] = v.upcase }
```

O anche la mappa :

1.8

```
new_fruit = Hash[fruit.map{ |k,v| [k.to_s, v.upcase] }]
```

(Vedi questa risposta per maggiori dettagli, incluso come manipolare gli hash sul posto.)

## Iterating Over a Hash

Un `Hash` include il modulo `Enumerable` , che fornisce diversi metodi di iterazione, come: `Enumerable#each` , `Enumerable#each_pair` , `Enumerable#each_key` e `Enumerable#each_value` .

`.each` e `.each_pair` iterano su ciascuna coppia chiave-valore:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each do |key, value|
    puts "#{key} = #{value}"
end

# => first_name = John
#    last_name = Doe
```

`.each_key` iterazioni solo sulle chiavi:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_key do |key|
  puts key
end

# => first_name
#    last_name
```

`.each_value` iterazioni solo sui valori:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_value do |value|
    puts value
end

# => John
#    Doe
```

`.each_with_index` iterazioni sugli elementi e fornisce l'indice dell'iterazione:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_with_index do |(key, value), index|
    puts "index: #{index} | key: #{key} | value: #{value}"
end

# => index: 0 | key: first_name | value: John
#    index: 1 | key: last_name | value: Doe
```

## Conversione da e verso le matrici

Gli hash possono essere convertiti liberamente da e verso gli array. La conversione di un hash di coppie chiave / valore in una matrice produrrà un array contenente array annidati per la coppia:

```
{ :a => 1, :b => 2 }.to_a # => [[:a, 1], [:b, 2]]
```

Nella direzione opposta, è possibile creare un hash da una matrice dello stesso formato:

```
[[:x, 3], [:y, 4]].to_h # => { :x => 3, :y => 4 }
```

Allo stesso modo, gli hash possono essere inizializzati usando `Hash[]` e un elenco di chiavi e valori alternati:

```
Hash[:a, 1, :b, 2] # => { :a => 1, :b => 2 }
```

O da una matrice di matrici con due valori ciascuno:

```
Hash[ [[:x, 3], [:y, 4]] ] # => { :x => 3, :y => 4 }
```

Gli hash possono essere riconvertiti in una matrice di chiavi e valori alternati usando `flatten()` :

```
{ :a => 1, :b => 2 }.flatten # => [:a, 1, :b, 2]
```

La semplice conversione da e verso un array consente a `Hash` di funzionare bene con molti metodi `Enumerable` come `collect` e `zip` :

```
Hash[('a'..'z').collect{ |c| [c, c.upcase] }] # => { 'a' => 'A', 'b' => 'B', ... }

people = ['Alice', 'Bob', 'Eve']
height = [5.7, 6.0, 4.9]
Hash[people.zip(height)] # => { 'Alice' => 5.7, 'Bob' => '6.0', 'Eve' => 4.9 }
```

## Ottenere tutte le chiavi o valori di hash

```
{foo: 'bar', biz: 'baz'}.keys   # => [:foo, :biz]
{foo: 'bar', biz: 'baz'}.values # => ["bar", "baz"]
{foo: 'bar', biz: 'baz'}.to_a   # => [[:foo, "bar"], [:biz, "baz"]]
{foo: 'bar', biz: 'baz'}.each   #<Enumerator: {:foo=>"bar", :biz=>"baz"}:each>
```

## Sovrascrivere la funzione di hash

Gli hash di Ruby usano i metodi `hash` ed `eql?` per eseguire l'operazione di hash e assegnare gli oggetti memorizzati nell'hash ai contenitori interni di hash. L'implementazione predefinita `hash` in Ruby è la funzione di hashing del soffio su tutti i campi membri dell'oggetto hash . Per sovrascrivere questo comportamento è possibile sovrascrivere `hash` ed `eql?` metodi.

Come con altre implementazioni di hash, due oggetti a e b saranno sottoposti a hashing sullo stesso bucket se `a.hash == b.hash` e saranno considerati identici se `a.eql?(b)` . Quindi, quando si reimplementa `hash` ed `eql?` si dovrebbe fare attenzione a garantire che `a` e `b` siano uguali sotto `eql?` devono restituire lo stesso valore di `hash` . Altrimenti ciò potrebbe causare voci duplicate in un hash. Viceversa, una scelta sbagliata nell'implementazione `hash` potrebbe portare molti oggetti a condividere lo stesso bucket hash, distruggendo efficacemente il tempo di ricerca O (1) e

causando O (n) per chiamare `eql?` su tutti gli oggetti.

Nell'esempio seguente solo l'istanza della classe `A` è memorizzata come chiave, poiché è stata aggiunta per prima:

```
class A
  def initialize(hash_value)
    @hash_value = hash_value
  end
  def hash
    @hash_value # Return the value given externally
  end
  def eql?(b)
    self.hash == b.hash
  end
end

class B < A
end

a = A.new(1)
b = B.new(1)

h = {}
h[a] = 1
h[b] = 2

raise "error" unless h.size == 1
raise "error" unless h.include? b
raise "error" unless h.include? a
```

## Filtro degli hash

`select` restituisce un nuovo `hash` con coppie chiave-valore per le quali il blocco restituisce `true` .

```
{ :a => 1, :b => 2, :c => 3 }.select { |k, v| k != :a && v.even? } # => { :b => 2 }
```

Quando non avrai bisogno della *chiave* o del *valore* in un blocco filtro, la convenzione è di usare _ in quel punto:

```
{ :a => 1, :b => 2, :c => 3 }.select { |_, v| v.even? } # => { :b => 2 }
{ :a => 1, :b => 2, :c => 3 }.select { |k, _| k == :c } # => { :c => 3 }
```

`reject` un nuovo `hash` con coppie chiave-valore per le quali il blocco restituisce `false` :

```
{ :a => 1, :b => 2, :c => 3 }.reject { |_, v| v.even? } # => { :a => 1, :c => 3 }
{ :a => 1, :b => 2, :c => 3 }.reject { |k, _| k == :b } # => { :a => 1, :c => 3 }
```

## Imposta le operazioni sugli hash

- **Intersezione di Hash**

  Per ottenere l'intersezione di due hash, restituisci le chiavi condivise i cui valori sono uguali:

```
hash1 = { :a => 1, :b => 2 }
hash2 = { :b => 2, :c => 3 }
hash1.select { |k, v| (hash2.include?(k) && hash2[k] == v) } # => { :b => 2 }
```

- **Unione (unione) di hash:**

  le chiavi in un hash sono univoche, se una chiave si verifica in entrambi gli hash che devono essere uniti, quello dell'hash con cui viene eseguita l' `merge` viene sovrascritto:

```
hash1 = { :a => 1, :b => 2 }
hash2 = { :b => 4, :c => 3 }

hash1.merge(hash2) # => { :a => 1, :b => 4, :c => 3 }
hash2.merge(hash1) # => { :b => 2, :c => 3, :a => 1 }
```

Leggi hash online: https://riptutorial.com/it/ruby/topic/288/hash

# Capitolo 33: Iniziare con Hanami

## introduzione

La mia missione qui è di contribuire con la comunità per aiutare le nuove persone che vogliono conoscere questa fantastica struttura - Hanami.

**Ma come funzionerà?**

Esercitazioni brevi e semplici che mostrano esempi di Hanami e seguendo i prossimi tutorial vedremo come testare la nostra applicazione e creare una semplice API REST.

**Iniziamo!**

## Examples

### A proposito di Hanami

Oltre ad Hanami, una struttura leggera e veloce, uno dei punti su cui la maggior parte richiama l'attenzione è il concetto di **architettura pulita in** cui ci mostra che la struttura non è la nostra applicazione, come ha detto Robert Martin in precedenza.

Il design dell'arredamento Hanami ci offre l'uso del **Container** , in ogni Container abbiamo la nostra applicazione indipendentemente dal framework. Ciò significa che possiamo prendere il nostro codice e inserirlo in un framework Rails, ad esempio.

**Hanami è un framework MVC?**

L'idea dei framework MVC è di costruire una struttura seguendo il modello -> Controller -> Visualizza. Hanami segue il Modello | Controller -> Visualizza -> Modello. Il risultato è un'applicazione più incolore, seguendo i principi **SOLID** e molto più pulita.

**- Link importanti.**

Hanami http://hanamirb.org/

Robert Martin - Clean Arquitecture https://www.youtube.com/watch?v=WpkDN78P884

Clean Arquitecture https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Principi SOLIDI http://practicingruby.com/articles/solid-design-principles

### Come installare Hanami?

- **Passaggio 1:** installazione della gemma Hanami.

```
$ gem install hanami
```

- **Passo 2** : Genera un nuovo progetto impostando RSpec come framework di test.

  Apri una riga di comando o un terminale. Per generare una nuova applicazione hanami, usa hanami new seguito dal nome della tua app e dal parametro test rspec.

```
$ hanami new "myapp" --test=rspec
```

Obs. Per impostazione predefinita, Hanami imposta Minitest come framework di test.

Questo creerà un'applicazione hanami chiamata myapp in una directory myapp e installerà le dipendenze gem che sono già menzionate in Gemfile usando l'installazione bundle.

Per passare a questa directory, utilizzare il comando cd, che sta per change directory.

```
$ cd my_app
$ bundle install
```

La directory myapp ha un numero di file e cartelle generati automaticamente che costituiscono la struttura di un'applicazione Hanami. Di seguito è riportato un elenco di file e cartelle che vengono creati per impostazione predefinita:

- **Gemfile** definisce le nostre dipendenze Rubygems (usando Bundler).
- **Rakefile** descrive i nostri compiti di Rake.
- **le app** contengono una o più applicazioni Web compatibili con Rack. Qui possiamo trovare la prima applicazione Hanami generata chiamata Web. È il luogo in cui troviamo i nostri controller, viste, percorsi e modelli.
- **config** contiene i file di configurazione.
- **config.ru** è per server rack.
- **db** contiene il nostro schema di database e migrazioni.
- **lib** contiene la nostra logica aziendale e il modello di dominio, incluse entità e repository.
- **il pubblico** conterrà le risorse statiche compilate.
- **le specifiche** contengono i nostri test.
- **Collegamenti importanti**

Gemma di Hanami https://github.com/hanami/hanami

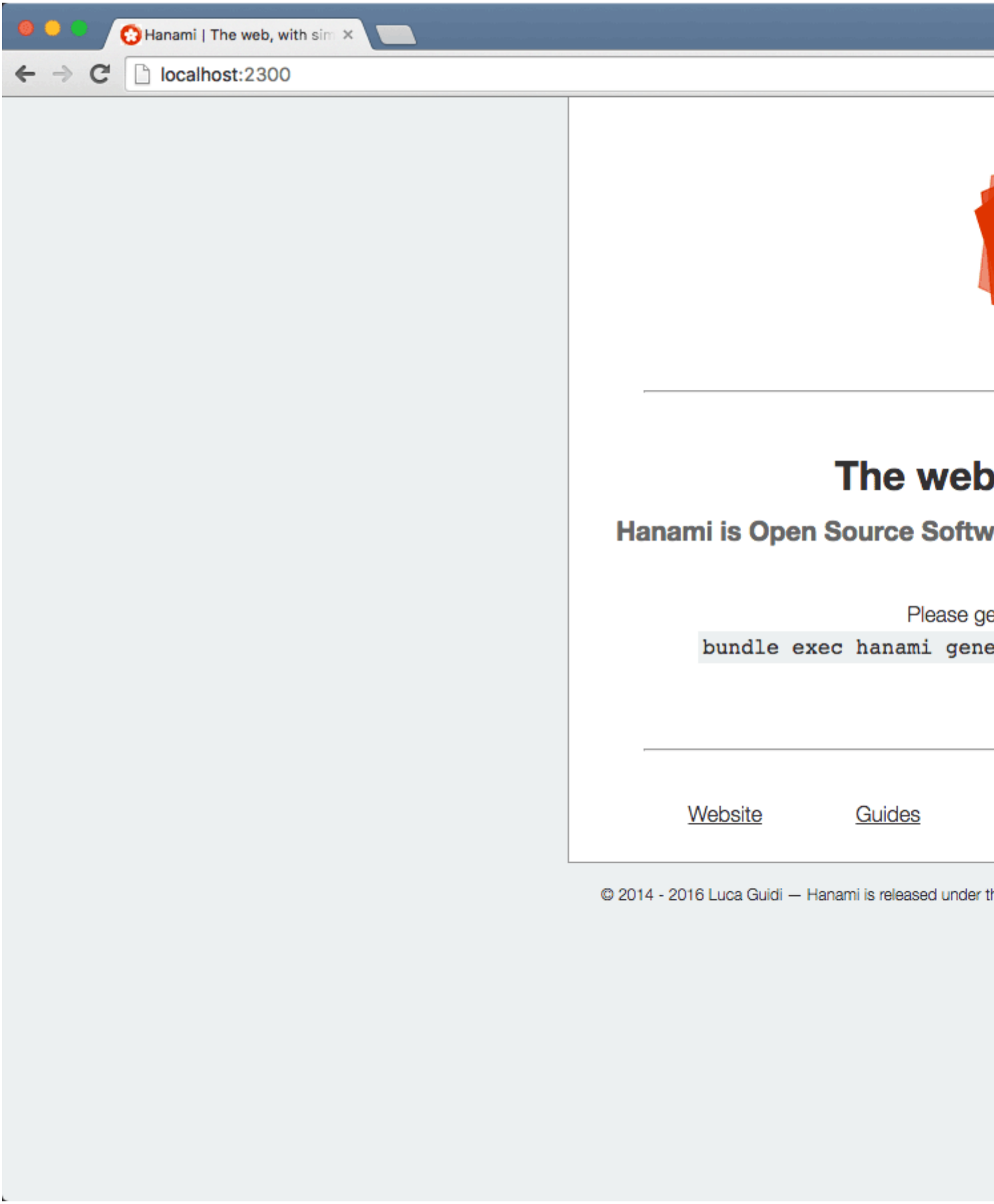
Guida introduttiva ufficiale di Hanami http://hanamirb.org/guides/getting-started/

## Come avviare il server?

- **Passo 1:** Per avviare il server basta digitare il comando qui sotto, poi vedrai la pagina

iniziale.

```
$ bundle exec hanami server
```



Leggi Iniziare con Hanami online: https://riptutorial.com/it/ruby/topic/9676/iniziare-con-hanami

# Capitolo 34: Installazione

## Examples

### Linux - Compilando dalla fonte

`In questo modo otterrai il rubino più nuovo ma ha i suoi lati negativi. Farlo come questo rubino non sarà gestito da nessuna applicazione.

**!! Ricorda di chagne la versione in modo che corrisponda al tuo !!**

1. è necessario scaricare un tarball trovare un collegamento su un sito web ufficiale ( https://www.ruby-lang.org/en/downloads/)
2. Estrai il tarball
3. Installare

```
$> wget https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.3.tar.gz
$> tar -xvzf ruby-2.3.3.tar.gz
$> cd ruby-2.3.3
$> ./configure
$> make
$> sudo make install
```

Questo installerà ruby in `/usr/local` . Se non si è soddisfatti di questa posizione, è possibile passare un argomento a `./configure --prefix=DIR` dove `DIR` è la directory in cui si desidera installare Ruby.

### Installazione Linux usando un gestore di pacchetti

Probabilmente la scelta più semplice, ma attenzione, la versione non è sempre la più recente. Basta aprire il terminale e digitare (a seconda della distribuzione)

in Debian o Ubuntu usando apt

```
$> sudo apt install ruby
```

in CentOS, openSUSE o Fedora

```
$> sudo yum install ruby
```

È possibile utilizzare l'opzione `-y` modo che non venga richiesto di concordare con l'installazione, ma a mio parere è buona norma controllare sempre che cosa sta tentando di installare il gestore pacchetti.

### Windows: installazione tramite il programma di installazione

Probabilmente il modo più semplice per impostare ruby su Windows è andare su

http://rubyinstaller.org/ e da lì scaricare un file eseguibile che installerai.

Non devi impostare quasi nulla, ma ci sarà una finestra importante. Avrà una casella di controllo che dice *Aggiungi ruby eseguibile al tuo PERCORSO* . Confermare che è **selezionato** , se non lo si controlla, altrimenti non sarà possibile eseguire ruby e dovrà impostare la variabile PATH da solo.

Quindi vai avanti fino a quando non si installa e questo è quello.

### Gems

In questo esempio useremo "nokogiri" come gemma di esempio. 'nokogiri' può in seguito essere sostituito da qualsiasi altro nome di gemma.

Per lavorare con le gemme utilizziamo uno strumento da riga di comando chiamato `gem` seguito da un'opzione come `install` o `update` e quindi i nomi delle gemme che vogliamo installare, ma non è tutto.

Installa gemme:

```
$> gem install nokogiri
```

Ma non è l'unica cosa di cui abbiamo bisogno. Possiamo anche specificare la versione, l'origine da cui installare o cercare le gemme. Iniziamo con alcuni casi d'uso di base (UC) e in seguito è possibile richiedere un aggiornamento.

Elenco di tutte le gemme installate:

```
$> gem list
```

Disinstallazione di gemme:

```
$> gem uninstall nokogiri
```

Se avremo più versioni della gemma nokogiri, ti verrà richiesto di specificare quale vogliamo disinstallare. Otterremo una lista ordinata e numerata e scriviamo semplicemente il numero.

Aggiornamento delle gemme

```
$> gem update nokogiri
```

o se vogliamo aggiornarli tutti

```
$> gem update
```

La `gem` Comman ha molti altri usi e opzioni da esplorare. Per ulteriori informazioni, consultare la documentazione ufficiale. Se qualcosa non è chiaro, invia una richiesta e io la aggiungerò.

## Linux - risoluzione dei problemi di installazione gem

Primo UC nell'esempio **Gems** `$> gem install nokogiri` può avere un problema nell'installazione di gemme perché non abbiamo i permessi per questo. Questo può essere risolto in più di un solo modo.

Prima soluzione UC a:

U puoi usare `sudo` . Questo installerà la gemma per tutti gli utenti. Questo metodo dovrebbe essere disapprovato. Questo dovrebbe essere usato solo con la gemma che sai sarà utilizzabile da tutti gli utenti. Solitamente nella vita reale non vuoi che qualche utente abbia accesso a `sudo` .

```
$> sudo gem install nokogiri
```

Prima soluzione UC b

U può usare l'opzione `--user-install` che installa le gemme nella cartella gem degli utenti (solitamente in `~/.gem` )

```
&> gem install nokogiri --user-install
```

Prima soluzione UC c

U può impostare GEM_HOME e GEM_PATH, quindi eseguirà il comando `gem install` install tutte le gemme in una cartella specificata dall'utente. Posso darti un esempio di ciò (nel solito modo)

- Prima di tutto è necessario aprire .bashrc. Usa nano o il tuo editor di testo preferito.

```
$> nano ~/.bashrc
```

- Quindi alla fine di questo file scrivi

```
export GEM_HOME=$HOME/.gem
export GEM_PATH=$HOME/.gem
```

- Ora dovrai riavviare il terminale o scrivere . `~/.bashrc` per ricaricare la configurazione. Questo ti permetterà di usare `gem isntall nokogiri` e installerà queste gemme nella cartella che hai specificato.

## Installazione di Ruby macOS

Quindi la buona notizia è che Apple include gentilmente un interprete Ruby. Sfortunatamente, tende a non essere una versione recente:

```
$ /usr/bin/ruby -v
ruby 2.0.0p648 (2015-12-16 revision 53162) [universal.x86_64-darwin16]
```

Se hai installato Homebrew , puoi ottenere l'ultimo Ruby con:

```
$ brew install ruby

$ /usr/local/bin/ruby -v
ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]
```

(Probabilmente vedrai una versione più recente se la proverai).

Per raccogliere la versione prodotta senza utilizzare il percorso completo, ti consigliamo di aggiungere `/usr/local/bin` all'inizio della tua variabile d'ambiente `$PATH` :

```
export PATH=/usr/local/bin:$PATH
```

L'aggiunta di quella riga a `~/.bash_profile` garantisce che questa versione sarà disponibile dopo il riavvio del sistema:

```
$ type ruby
ruby is /usr/local/bin/ruby
```

Homebrew installerà `gem` per l' installazione di gemme . È anche possibile costruire dalla fonte se necessario. Homebrew include anche questa opzione:

```
$ brew install ruby --build-from-source
```

Leggi Installazione online: https://riptutorial.com/it/ruby/topic/8095/installazione

# Capitolo 35: instance_eval

## Sintassi

- object.instance_eval 'code'
- object.instance_eval 'code', 'filename'
- object.instance_eval 'code', 'filename', 'numero di riga'
- object.instance_eval {code}
- object.instance_eval {| receiver | codice }

## Parametri

| Parametro | Dettagli |
|---|---|
| `string` | Contiene il codice sorgente di Ruby da valutare. |
| `filename` | Nome del file da utilizzare per la segnalazione degli errori. |
| `lineno` | Numero di riga da utilizzare per la segnalazione degli errori. |
| `block` | Il blocco di codice da valutare. |
| `obj` | Il ricevitore viene passato al blocco come unico argomento. |

## Examples

### Valutazione delle istanze

Il metodo `instance_eval` è disponibile su tutti gli oggetti. Valuta il codice nel contesto del destinatario:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable # => :value
```

`instance_eval` imposta `self` per `object` per la durata del blocco di codice:

```
object.instance_eval { self == object } # => true
```

Il destinatario viene anche passato al blocco come unico argomento:

```
object.instance_eval { |argument| argument == object } # => true
```

Il metodo `instance_exec` differisce in questo senso: passa invece i suoi argomenti al blocco.

```
object.instance_exec :@variable do |name|
  instance_variable_get name # => :value
end
```

## Implementazione con

Molte lingue sono dotate di `with` dichiarazione che consente ai programmatori di omettere il ricevitore di chiamate di metodo.

`with` può essere facilmente emulato in Ruby usando `instance_eval` :

```
def with(object, &block)
  object.instance_eval &block
end
```

Il metodo `with` può essere utilizzato per eseguire senza problemi metodi sugli oggetti:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Leggi instance_eval online: https://riptutorial.com/it/ruby/topic/5049/instance-eval

# Capitolo 36: Introspezione

## Examples

### Visualizza i metodi di un oggetto

## Ispezionare un oggetto

È possibile trovare i metodi pubblici a cui un oggetto può rispondere utilizzando i `methods` oi metodi `public_methods` , che restituiscono una matrice di simboli:

```
class Foo
  def bar; 42; end
end
f = Foo.new
def f.yay; 17; end
p f.methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class, :clone,
#=>  :define_singleton_method, :display, :dup, :enum_for, :eql?, :equal?, :extend,
#=>  :freeze, :frozen?, :hash, :inspect, :instance_eval, :instance_exec,
#=>  :instance_of?, :instance_variable_defined?, :instance_variable_get,
#=>  :instance_variable_set, :instance_variables, :is_a?, :itself, :kind_of?,
#=>  :method, :methods, :nil?, :object_id, :private_methods, :protected_methods,
#=>  :public_method, :public_methods, :public_send, :remove_instance_variable,
#=>  :respond_to?, :send, :singleton_class, :singleton_method, :singleton_methods,
#=>  :taint, :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust,
#=>  :untrusted?, :yay]
```

Per un elenco più mirato, è possibile rimuovere i metodi comuni a tutti gli oggetti, ad es

```
p (f.methods - Object.methods).sort
#=> [:bar,:yay]
```

In alternativa, puoi passare `false` a `methods` o `public_methods` :

```
p f.methods(false) # public and protected singleton methods of `f`
#=> [:yay]

p f.public_methods(false)
#=> [:yay, :bar]
```

Puoi trovare i metodi privati e protetti di un oggetto usando `private_methods` e `protected_methods` :

```
p f.private_methods.sort
#=> [:Array, :Complex, :DelegateClass, :Float, :Hash, :Integer, :Rational, :String,
#=>  :__callee__, :__dir__, :__method__, :`, :abort, :at_exit, :autoload, :autoload?,
#=>  :binding, :block_given?, :caller, :caller_locations, :catch,
#=>  :default_src_encoding, :eval, :exec, :exit, :exit!, :fail, :fork, :format, :gem,
#=>  :gem_original_require, :gets, :global_variables, :initialize, :initialize_clone,
#=>  :initialize_copy, :initialize_dup, :irb_binding, :iterator?, :lambda, :load,
#=>  :local_variables, :loop, :method_missing, :open, :p, :print, :printf, :proc,
```

```
#=>  :putc, :puts, :raise, :rand, :readline, :readlines, :require, :require_relative,
#=>  :respond_to_missing?, :select, :set_trace_func, :singleton_method_added,
#=>  :singleton_method_removed, :singleton_method_undefined, :sleep, :spawn,
#=>  :sprintf, :srand, :syscall, :system, :test, :throw, :trace_var, :trap,
#=>  :untrace_var, :warn]

p f.protected_methods
#=> []
```

Come per i `methods` e `public_methods` , puoi passare `false` a `private_methods` e `protected_methods` per tagliare via i metodi ereditati.

# Ispezionando una classe o un modulo

Oltre ai `methods` , `public_methods` , `protected_methods` e `private_methods` , classi e moduli espongono `instance_methods` , `public_instance_methods` , `protected_instance_methods` e `private_instance_methods` per determinare i metodi esposti per gli oggetti che ereditano dalla classe o dal modulo. Come sopra, puoi passare `false` a questi metodi per escludere metodi ereditati:

```
p Foo.instance_methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class,
#=>  :clone, :define_singleton_method, :display, :dup, :enum_for, :eql?,
#=>  :equal?, :extend, :freeze, :frozen?, :hash, :inspect, :instance_eval,
#=>  :instance_exec, :instance_of?, :instance_variable_defined?,
#=>  :instance_variable_get, :instance_variable_set, :instance_variables,
#=>  :is_a?, :itself, :kind_of?, :method, :methods, :nil?, :object_id,
#=>  :private_methods, :protected_methods, :public_method, :public_methods,
#=>  :public_send, :remove_instance_variable, :respond_to?, :send,
#=>  :singleton_class, :singleton_method, :singleton_methods, :taint,
#=>  :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust, :untrusted?]

p Foo.instance_methods(false)
#=> [:bar]
```

Infine, se dimentichi i nomi di molti di questi in futuro, puoi trovare tutti questi metodi usando i `methods` :

```
p f.methods.grep(/methods/)
#=> [:private_methods, :methods, :protected_methods, :public_methods,
#=>  :singleton_methods]

p Foo.methods.grep(/methods/)
#=> [:public_instance_methods, :instance_methods, :private_instance_methods,
#=>  :protected_instance_methods, :private_methods, :methods,
#=>  :protected_methods, :public_methods, :singleton_methods]
```

### Visualizza le variabili di istanza di un oggetto

È possibile interrogare un oggetto sulle sue variabili di `instance_variables` usando `instance_variables` , `instance_variable_defined?` e `instance_variable_get` e modificarli usando `instance_variable_set` e `remove_instance_variable` :

```
class Foo
  attr_reader :bar
  def initialize
    @bar = 42
  end
end
f = Foo.new
f.instance_variables                 #=> [:@bar]
f.instance_variable_defined?(:@baz) #=> false
f.instance_variable_defined?(:@bar) #=> true
f.instance_variable_get(:@bar)       #=> 42
f.instance_variable_set(:@bar, 17)    #=> 17
f.bar                                #=> 17
f.remove_instance_variable(:@bar)    #=> 17
f.bar                                #=> nil
f.instance_variables                 #=> []
```

I nomi delle variabili di istanza includono il simbolo `@` . Riceverai un errore se lo ometti:

```
f.instance_variable_defined?(:jim)
#=> NameError: `jim' is not allowed as an instance variable name
```

## Visualizza variabili globali e locali

Il `Kernel` espone i metodi per ottenere l'elenco di `global_variables` e `local_variables` :

```
cats  = 42
$demo = "in progress"
p global_variables.sort
#=> [:$!, :$", :$$, :$&, :$', :$*, :$+, :$,, :$-0, :$-F, :$-I, :$-K, :$-W, :$-a,
#=>  :$-d, :$-i, :$-l, :$-p, :$-v, :$-w, :$., :$/, :$0, :$1, :$2, :$3, :$4, :$5,
#=>  :$6, :$7, :$8, :$9, :$:, :$;, :$<, :$=, :$>, :$?, :$@, :$DEBUG, :$FILENAME,
#=>  :$KCODE, :$LOADED_FEATURES, :$LOAD_PATH, :$PROGRAM_NAME, :$SAFE, :$VERBOSE,
#=>  :$\, :$_, :$`, :$binding, :$demo, :$stderr, :$stdin, :$stdout, :$~]

p local_variables
#=> [:cats]
```

A differenza delle variabili di istanza, non esistono metodi specifici per ottenere, impostare o rimuovere variabili globali o locali. La ricerca di tale funzionalità è di solito un segno che il tuo codice dovrebbe essere riscritto per utilizzare un hash per memorizzare i valori. Tuttavia, se è necessario modificare le variabili globali o locali per nome, è possibile utilizzare `eval` con una stringa:

```
var = "$demo"
eval(var)           #=> "in progress"
eval("#{var} = 17")
p $demo             #=> 17
```

Di default, `eval` valuterà le tue variabili nell'ambito corrente. Per valutare le variabili locali in un ambito diverso, è necessario acquisire il *legame in* cui esistono le variabili locali.

```
def local_variable_get(name, bound=nil)
```

```
  foo = :inside
  eval(name,bound)
end

def test_1
  foo = :outside
  p local_variable_get("foo")
end

def test_2
  foo = :outside
  p local_variable_get("foo",binding)
end

test_1 #=> :inside
test_2 #=> :outside
```

In quanto sopra, `test_1` non ha passato un'associazione a `local_variable_get` , e quindi l' `eval` stato eseguito nel contesto di quel metodo, dove una variabile locale di nome `foo` stata impostata su `:inside` .

## Visualizza le variabili di classe

Classi e moduli hanno gli stessi metodi per l'analisi delle variabili di istanza come qualsiasi altro oggetto. Anche la classe e i moduli hanno metodi simili per interrogare le variabili di classe ( `@@these_things` ):

```
p Module.methods.grep(/class_variable/)
#=> [:class_variables, :class_variable_get, :remove_class_variable,
#=>  :class_variable_defined?, :class_variable_set]

class Foo
  @@instances = 0
  def initialize
    @@instances += 1
  end
end

class Bar < Foo; end

5.times{ Foo.new }
3.times{ Bar.new }
p Foo.class_variables                    #=> [:@@instances]
p Bar.class_variables                    #=> [:@@instances]
p Foo.class_variable_get(:@@instances)   #=> 8
p Bar.class_variable_get(:@@instances)   #=> 8
```

Simile alle variabili di istanza, il nome delle variabili di classe deve iniziare con `@@` oppure si otterrà un errore:

```
p Bar.class_variable_defined?( :instances )
#=> NameError: `instances' is not allowed as a class variable name
```

Leggi Introspezione online: https://riptutorial.com/it/ruby/topic/6227/introspezione

# Capitolo 37: Introspezione in Ruby

## introduzione

**Cos'è l'introspezione?**

L'introspezione guarda dentro per sapere dell'interno. Questa è una semplice definizione di introspezione.

Nella programmazione e in Ruby in generale ... l'introspezione è la capacità di guardare l'oggetto, la classe ... in fase di esecuzione per sapere di quello.

## Examples

### Vediamo alcuni esempi

Esempio:

```
s = "Hello"  # s is a string
```

Quindi scopriamo qualcosa su s. Cominciamo:

Quindi vuoi sapere qual è la classe di s in fase di esecuzione?

```
irb(main):055:0* s.class
=> String
```

Ohh, bene. Ma quali sono i metodi di s?

```
irb(main):002:0> s.methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*,
:+, :count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize,
:match, :succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte,
:getbyte, :<=>, :<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=,
:upcase, :downcase!, :capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars,
:split, :capitalize!, :swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend,
:scan, :ord, :reverse!, :center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop,
:crypt, :gsub!, :start_with?, :rstrip, :sub!, :ljust, :length, :size, :strip!, :succ,
:rstrip!, :chomp, :strip, :rjust, :lstrip!, :tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str,
:to_sym, :chop!, :each_byte, :each_char, :each_codepoint, :to_s, :to_i, :tr_s, :delete,
:encoding, :force_encoding, :sum, :delete!, :squeeze!, :tr, :to_f, :valid_encoding?, :slice,
:slice!, :rpartition, :each_line, :b, :ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?,
:instance_of?, :public_send, :instance_variable_get, :instance_variable_set,
:instance_variable_defined?, :remove_instance_variable, :private_methods, :kind_of?,
:instance_variables, :tap, :is_a?, :extend, :to_enum, :enum_for, :!~, :respond_to?, :display,
:object_id, :send, :method, :public_method, :singleton_method, :define_singleton_method,
:nil?, :class, :singleton_class, :clone, :dup, :itself, :taint, :tainted?, :untaint, :untrust,
:trust, :untrusted?, :methods, :protected_methods, :frozen?, :public_methods,
:singleton_methods, :!, :!=, :__send__, :equal?, :instance_eval, :instance_exec, :__id__]
```

Vuoi sapere se s è un'istanza di String?

```
irb(main):017:0*
irb(main):018:0* s.instance_of?(String)
=> true
```

Quali sono i metodi pubblici di s?

```
irb(main):026:0* s.public_methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*,
:+, :count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize,
:match, :succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte,
:getbyte, :<=>, :<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=,
:upcase, :downcase!, :capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars,
:split, :capitalize!, :swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend,
:scan, :ord, :reverse!, :center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop,
:crypt, :gsub!, :start_with?, :rstrip, :sub!, :ljust, :length, :size, :strip!, :succ,
:rstrip!, :chomp, :strip, :rjust, :lstrip!, :tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str,
:to_sym, :chop!, :each_byte, :each_char, :each_codepoint, :to_s, :to_i, :tr_s, :delete,
:encoding, :force_encoding, :sum, :delete!, :squeeze!, :tr, :to_f, :valid_encoding?, :slice,
:slice!, :rpartition, :each_line, :b, :ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?,
:pretty_print, :pretty_print_cycle, :pretty_print_instance_variables, :pretty_print_inspect,
:instance_of?, :public_send, :instance_variable_get, :instance_variable_set,
:instance_variable_defined?, :remove_instance_variable, :private_methods, :kind_of?,
:instance_variables, :tap, :pretty_inspect, :is_a?, :extend, :to_enum, :enum_for, :!~,
:respond_to?, :display, :object_id, :send, :method, :public_method, :singleton_method,
:define_singleton_method, :nil?, :class, :singleton_class, :clone, :dup, :itself, :taint,
:tainted?, :untaint, :untrust, :trust, :untrusted?, :methods, :protected_methods, :frozen?,
:public_methods, :singleton_methods, :!, :!=, :__send__, :equal?, :instance_eval,
:instance_exec, :__id__]
```

e metodi privati ....

```
irb(main):030:0* s.private_methods
=> [:initialize, :initialize_copy, :DelegateClass, :default_src_encoding, :irb_binding,
:sprintf, :format, :Integer, :Float, :String, :Array, :Hash, :catch, :throw, :loop,
:block_given?, :Complex, :set_trace_func, :trace_var, :untrace_var, :at_exit, :Rational,
:caller, :caller_locations, :select, :test, :fork, :exit, :`, :gem_original_require, :sleep,
:pp, :respond_to_missing?, :load, :exec, :exit!, :system, :spawn, :abort, :syscall, :printf,
:open, :putc, :print, :readline, :puts, :p, :srand, :readlines, :gets, :rand, :proc, :lambda,
:trap, :initialize_clone, :initialize_dup, :gem, :require, :require_relative, :autoload,
:autoload?, :binding, :local_variables, :warn, :raise, :fail, :global_variables, :__method__,
:__callee__, :__dir__, :eval, :iterator?, :method_missing, :singleton_method_added,
:singleton_method_removed, :singleton_method_undefined]
```

Sì, il nome del metodo è superiore. Vuoi ottenere la versione maiuscola di s? Proviamo:

```
irb(main):044:0> s.respond_to?(:upper)
=> false
```

Sembra che no, il metodo corretto è il controllo di upcase:

```
irb(main):047:0*
irb(main):048:0* s.respond_to?(:upcase)
=> true
```

## Introspezione di classe

Seguiamo la definizione della classe

```
class A
   def a; end
end

module B
   def b; end
end

class C < A
   include B
   def c; end
end
```

Quali sono i metodi di istanza di `C` ?

```
C.instance_methods # [:c, :b, :a, :to_json, :instance_of?...]
```

Quali sono i metodi di istanza che dichiarano solo su `C` ?

```
C.instance_methods(false) # [:c]
```

Quali sono gli antenati della classe `C` ?

```
C.ancestors # [C, B, A, Object,...]
```

Superclasse di `C` ?

```
C.superclass # A
```

Leggi Introspezione in Ruby online: https://riptutorial.com/it/ruby/topic/8752/introspezione-in-ruby

# Capitolo 38: IRB

## introduzione

IRB significa "Interactive Ruby Shell". Fondamentalmente ti permette di eseguire i comandi ruby in tempo reale (come fa la shell normale). IRB è uno strumento indispensabile quando si ha a che fare con l'API di Ruby. Funziona come uno script rb classico. Usalo per comandi brevi e facili. Una delle belle funzioni IRB è che quando premi la scheda mentre digiti un metodo ti darà un consiglio su cosa puoi usare (Questo non è un IntelliSense)

## Parametri

| Opzione | Dettagli |
|---|---|
| -f | Sopprimere la lettura di ~ / .irbrc |
| -m | Modalità Bc (carica matematica, frazione o matrice sono disponibili) |
| -d | Imposta $ DEBUG su true (come `ruby -d ') |
| -r modulo di carico | Lo stesso di `ruby -r ' |
| -Il percorso | Specifica la directory $ LOAD_PATH |
| -U | Uguale a `ruby -U` |
| -E enc | Come `ruby -E` |
| -w | Come `ruby -w` |
| -W [level = 2] | Come `ruby -W` |
| --ispezionare | Usa 'inspect' per l'output (default eccetto per la modalità bc) |
| --noinspect | Non usare ispezionare per la stampa |
| --linea di lettura | Utilizzare il modulo di estensione Readline |
| --noreadline | Non utilizzare il modulo di estensione Readline |
| --prompt prompt-mode | Passa alla modalità prompt. Le modalità prompt `default',` sono `default',` simple', `xmp' and` inf-ruby' |
| --inf-ruby-mode | Usa prompt appropriato per inf-ruby-mode su emacs. Sopprime --readline. |
| --simple prompt | Modalità prompt semplice |

| Opzione | Dettagli |
|---|---|
| --noprompt | Nessuna modalità prompt |
| --tracer | Mostra traccia per ogni esecuzione di comandi. |
| --back-trace-limit n | Visualizza backtrace top n e tail n. Il valore predefinito è 16. |
| --irb_debug n | Imposta il livello di debug interno su n (non per uso popolare) |
| -v, --version | Stampa la versione di irb |

# Examples

## Uso di base

IRB significa "Interactive Ruby Shell", permettendoci di eseguire espressioni ruby dall'input standard.

Per iniziare, `irb` nella shell. Puoi scrivere qualsiasi cosa in Ruby, dalle espressioni semplici:

```
$ irb
2.1.4 :001 > 2+2
=> 4
```

a casi complessi come i metodi:

```
2.1.4 :001> def method
2.1.4 :002?>   puts "Hello World"
2.1.4 :003?> end
=> :method
2.1.4 :004 > method
Hello World
=> nil
```

## Avvio di una sessione IRB all'interno di uno script Ruby

A partire da Ruby 2.4.0, è possibile avviare una sessione IRB interattiva all'interno di qualsiasi script Ruby utilizzando queste linee:

```
require 'irb'
binding.irb
```

Ciò avvierà un REPL IBR in cui si avrà il valore previsto per `self` e sarà possibile accedere a tutte le variabili locali e alle variabili di istanza che rientrano nello scope. Premi Ctrl + D o `quit` per riprendere il tuo programma Ruby.

Questo può essere molto utile per il debug.

Leggi IRB online: https://riptutorial.com/it/ruby/topic/4800/irb

# Capitolo 39: Iterazione

## Examples

### Ogni

Ruby ha molti tipi di enumeratori ma il primo e più semplice tipo di enumeratore da cui iniziare è `each` . Faremo stampare `even` o `odd` per ogni numero compreso tra `1` e `10` per mostrare come `each` opera.

Fondamentalmente ci sono due modi per passare i cosiddetti `blocks` . Un `block` è un pezzo di codice che viene passato che verrà eseguito dal metodo chiamato. `each` metodo prende un `block` che chiama per ogni elemento della collezione di oggetti su cui è stato chiamato.

Esistono due modi per passare un blocco a un metodo:

## Metodo 1: in linea

```
(1..10).each { |i| puts i.even? ? 'even' : 'odd' }
```

Questo è un modo molto compresso e *rubino* per risolvere questo. Rompiamo questo pezzo per pezzo.

1. `(1..10)` è un intervallo compreso tra `1` e `10` inclusi. Se volessimo che fosse da `1` a `10` esclusivi, scriveremmo `(1...10)` .
2. `.each` è un enumeratore che enumera su `each` elemento nell'oggetto su cui sta agendo. In questo caso, agisce su `each` numero nell'intervallo.
3. `{ |i| puts i.even? ? 'even' : 'odd' }` è il blocco per `each` istruzione, che a sua volta può essere ulteriormente suddivisa.
    1. `|i|` questo significa che ogni elemento nell'intervallo è rappresentato all'interno del blocco dall'identificatore `i` .
    2. `puts` è un metodo di output in Ruby che ha un'interruzione di riga automatica ogni volta che viene stampata. (Possiamo usare la `print` se non vogliamo l'interruzione automatica della linea)
    3. `i.even?` controlla se `i` è pari. Avremmo potuto usare anche `i % 2 == 0` ; tuttavia, è preferibile utilizzare metodi incorporati.
    4. `? "even" : "odd"` questo è l'operatore ternario di Ruby. Il modo in cui un operatore ternario è costruito è `expression ? a : b` . Questo è l'abbreviazione di

    ```
    if expression
      a
    else
      b
    end
    ```

Per il codice più lungo di una riga, il `block` deve essere passato come un `multiline block` .

## Metodo 2: multilinea

```
(1..10).each do |i|
  if i.even?
    puts 'even'
  else
    puts 'odd'
  end
end
```

In un `multiline block` il `do` sostituisce la parentesi aperta e `end` sostituisce la parentesi chiusa dallo stile in `inline` .

Ruby supporta anche reverse_each. Ripeterà l'array all'indietro.

```
@arr = [1,2,3,4]
puts @arr.inspect # output is [1,2,3,4]

print "Reversed array elements["
@arr.reverse_each do |val|
        print " #{val} " # output is 4 3 2 1
end
print "]\n"
```

---

### Implementazione in una classe

`Enumerable` è il modulo più popolare in Ruby. Il suo scopo è quello di fornire metodi iterabili come la `map` , `select` , `reduce` , ecc. Le classi che usano `Enumerable` includono `Array` , `Hash` , `Range` . Per usarlo, devi `include Enumerable` e implementare `each` .

```
class NaturalNumbers
  include Enumerable

  def initialize(upper_limit)
    @upper_limit = upper_limit
  end

  def each(&block)
    0.upto(@upper_limit).each(&block)
  end
end

n = NaturalNumbers.new(6)

n.reduce(:+)                    # => 21
n.select(&:even?)               # => [0, 2, 4, 6]
n.map { |number| number ** 2 } # => [0, 1, 4, 9, 16, 25, 36]
```

### Carta geografica

Restituisce l'oggetto modificato, ma l'oggetto originale rimane com'era. Per esempio:

```
arr = [1, 2, 3]
arr.map { |i| i + 1 } # => [2, 3, 4]
arr # => [1, 2, 3]
```

`map!` cambia l'oggetto originale:

```
arr = [1, 2, 3]
arr.map! { |i| i + 1 } # => [2, 3, 4]
arr # => [2, 3, 4]
```

Nota: puoi anche utilizzare `collect` per fare la stessa cosa.

## Iterare su oggetti complessi

### Array

È possibile eseguire iterazioni su array annidati:

```
[[1, 2], [3, 4]].each { |(a, b)| p "a: #{ a }", "b: #{ b }" }
```

La seguente sintassi è consentita anche:

```
[[1, 2], [3, 4]].each { |a, b| "a: #{ a }", "b: #{ b }" }
```

Produrrà:

```
"a: 1"
"b: 2"
"a: 3"
"b: 4"
```

### hash

Puoi eseguire l'iterazione su coppie chiave-valore:

```
{a: 1, b: 2, c: 3}.each { |pair| p "pair: #{ pair }" }
```

Produrrà:

```
"pair: [:a, 1]"
"pair: [:b, 2]"
"pair: [:c, 3]"
```

Puoi scorrere simultaneamente su chiavi e valori:

```
{a: 1, b: 2, c: 3}.each { |(k, v)| p "k: #{ k }", "v: #{ k }" }
```

Produrrà:

```
"k: a"
"v: a"
"k: b"
"v: b"
"k: c"
"v: c"
```

### Per l'iteratore

Questo itera da 4 a 13 (inclusi).

```
for i in 4..13
    puts "this is #{i}.th number"
end
```

Possiamo anche eseguire iterazioni su array usando for

```
names = ['Siva', 'Charan', 'Naresh', 'Manish']

for name in names
    puts name
end
```

### Iterazione con indice

A volte si desidera conoscere la posizione ( **indice** ) dell'elemento corrente mentre si esegue l'iterazione su un enumeratore. A tale scopo, Ruby fornisce il metodo `with_index` . Può essere applicato a tutti gli enumeratori. Fondamentalmente, aggiungendo `with_index` a un'enumerazione, è possibile enumerare tale enumerazione. L'indice viene passato a un blocco come secondo argomento.

```
[2,3,4].map.with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
#Element of array number 2 => 4
#=> [nil, nil, nil]
```

`with_index` ha un argomento opzionale - il primo indice che è `0` per impostazione predefinita:

```
[2,3,4].map.with_index(1) { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 1 => 2
#Element of array number 2 => 3
#Element of array number 3 => 4
#=> [nil, nil, nil]
```

Esiste un metodo specifico `each_with_index` . L'unica differenza tra esso e `each.with_index` è che non puoi passare un argomento a questo, quindi il primo indice è sempre `0` .

```
[2,3,4].each_with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
```

```
#Element of array number 2 => 4
#=> [2, 3, 4]
```

Leggi Iterazione online: https://riptutorial.com/it/ruby/topic/1159/iterazione

# Capitolo 40: JSON con Ruby

## Examples

### Usare JSON con Ruby

JSON (JavaScript Object Notation) è un formato di interscambio dati leggero. Molte applicazioni Web lo utilizzano per inviare e ricevere dati.

In Ruby puoi semplicemente lavorare con JSON.

All'inizio devi `require 'json'` , quindi puoi analizzare una stringa JSON tramite il comando `JSON.parse()` .

```
require 'json'

j = '{"a": 1, "b": 2}'
puts JSON.parse(j)
>> {"a"=>1, "b"=>2}
```

Quello che succede qui è che il parser genera un Ruby Hash dal JSON.

Al contrario, generare JSON con un hash di Ruby è semplice come l'analisi. Il metodo di scelta è `to_json` :

```
require 'json'

hash = { 'a' => 1, 'b' => 2 }
json = hash.to_json
puts json
>> {"a":1,"b":2}
```

### Utilizzo dei simboli

Puoi usare JSON insieme ai simboli Ruby. Con l'opzione symbolize_names per il parser, le chiavi dell'hash risultante saranno simboli anziché stringhe.

```
json = '{ "a": 1, "b": 2 }'
puts JSON.parse(json, symbolize_names: true)
>> {:a=>1, :b=>2}
```

Leggi JSON con Ruby online: https://riptutorial.com/it/ruby/topic/5853/json-con-ruby

# Capitolo 41: Messaggio in corso

## Examples

### introduzione

In *Object Oriented Design* , gli oggetti *ricevono* messaggi e *rispondono* a loro. In Ruby, l'invio di un messaggio *chiama un metodo* e il risultato di tale metodo è la risposta.

In Ruby il passaggio dei messaggi è dinamico. Quando arriva un messaggio invece di sapere esattamente come rispondere ad esso Ruby usa un insieme predefinito di regole per trovare un metodo che possa rispondere ad esso. Possiamo usare queste regole per interrompere e rispondere al messaggio, inviarlo a un altro oggetto o modificarlo tra le altre azioni.

Ogni volta che un oggetto riceve un messaggio, Ruby controlla:

1. Se questo oggetto ha una classe singleton e può rispondere a questo messaggio.
2. Cerca la classe di questo oggetto e la catena degli antenati di classe.
3. Uno ad uno controlla se un metodo è disponibile su questo antenato e si sposta verso l'alto della catena.

### Messaggio che passa attraverso la catena di ereditarietà

```
class Example
  def example_method
    :example
  end

  def subexample_method
    :example
  end

  def not_missed_method
    :example
  end

  def method_missing name
    return :example if name == :missing_example_method
    return :example if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

class SubExample < Example
  def subexample_method
    :subexample
  end

  def method_missing name
    return :subexample if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
```

```
    super
  end
end

s = Subexample.new
```

Per trovare un metodo adatto per `SubExample#subexample_method` Ruby guarda prima alla catena di antenati di `SubExample`

```
SubExample.ancestors # => [SubExample, Example, Object, Kernel, BasicObject]
```

Inizia da `SubExample` . Se inviamo il messaggio `subexample_method` , Ruby sceglie quello disponibile SubExample e ignora `Example#subexample_method` .

```
s.subexample_method # => :subexample
```

Dopo `SubExample` verifica `Example` . Se inviamo `example_method` Ruby controlla se `SubExample` può rispondere o no e poiché non può Ruby risalire la catena e guarda in `Example` .

```
s.example_method # => :example
```

Dopo che Ruby ha verificato tutti i metodi definiti, esegue `method_missing` per vedere se può rispondere o meno. Se inviamo `missing_subexample_method` Ruby non sarà in grado di trovare un metodo definito su `SubExample` quindi passa a `Example` . Non è possibile trovare un metodo definito su `Example` o qualsiasi altra classe più alta in catena. Ruby ricomincia ed esegue `method_missing` . `method_missing` di `SubExample` può rispondere a `missing_subexample_method` .

```
s.missing_subexample_method # => :subexample
```

Tuttavia, se viene definito un metodo, Ruby utilizza la versione definita anche se è più alta nella catena. Ad esempio, se inviamo `not_missed_method` anche se `method_missing` di `SubExample` può rispondere ad esso, Ruby si `SubExample` su `SubExample` perché non ha un metodo definito con quel nome e guarda in `Example` che ne ha uno.

```
s.not_missed_method # => :example
```

## Messaggio che passa attraverso la composizione del modulo

Ruby si sposta sulla catena di antenati di un oggetto. Questa catena può contenere sia moduli che classi. Le stesse regole su come spostare la catena si applicano anche ai moduli.

```
class Example
end

module Prepended
  def initialize *args
    return super :default if args.empty?
    super
  end
```

```
end

module FirstIncluded
  def foo
    :first
  end
end

module SecondIncluded
  def foo
    :second
  end
end

class SubExample < Example
  prepend Prepended
  include FirstIncluded
  include SecondIncluded

  def initialize data = :subexample
    puts data
  end
end

SubExample.ancestors # => [Prepended, SubExample, SecondIncluded, FirstIncluded, Example,
Object, Kernel, BasicObject]

s = SubExample.new # => :default
s.foo # => :second
```

## Interruzione dei messaggi

Esistono due modi per interrompere i messaggi.

- Utilizzare `method_missing` per interrompere qualsiasi messaggio non definito.
- Definire un metodo nel mezzo di una catena per intercettare il messaggio

Dopo aver interrotto i messaggi, è possibile:

- Rispondi a loro.
- Mandali da qualche altra parte.
- Modifica il messaggio o il suo risultato.

---

Interrompere tramite `method_missing` e rispondere al messaggio:

```
class Example
  def foo
    @foo
  end

  def method_missing name, data
    return super unless name.to_s =~ /=$/
    name = name.to_s.sub(/=$/, "")
    instance_variable_set "@#{name}", data
  end
end
```

```
e = Example.new

e.foo = :foo
e.foo # => :foo
```

---

Intercettare il messaggio e modificarlo:

```
class Example
  def initialize title, body
  end
end

class SubExample < Example
end
```

Ora immaginiamo che i nostri dati siano "title: body" e dobbiamo suddividerli prima di chiamare `Example` . Possiamo definire l' `initialize` su `SubExample` .

```
class SubExample < Example
  def initialize raw_data
    processed_data = raw_data.split ":"

    super processed_data[0], processed_data[1]
  end
end
```

---

Intercettazione del messaggio e invio a un altro oggetto:

```
class ObscureLogicProcessor
  def process data
    :ok
  end
end

class NormalLogicProcessor
  def process data
    :not_ok
  end
end

class WrapperProcessor < NormalLogicProcessor
  def process data
    return ObscureLogicProcessor.new.process data if data.obscure?

    super
  end
end
```

Leggi Messaggio in corso online: https://riptutorial.com/it/ruby/topic/5083/messaggio-in-corso

# Capitolo 42: metaprogrammazione

## introduzione

**La metaprogrammazione** può essere descritta in due modi:

"Programmi per computer che scrivono o manipolano altri programmi (o se stessi) come dati, o che fanno parte del lavoro in fase di compilazione che altrimenti verrebbero eseguiti in fase di esecuzione".

Più semplicemente: **Metaprogramming sta scrivendo un codice che scrive codice durante il runtime per semplificarti la vita** .

## Examples

### Implementare "con" usando la valutazione dell'istanza

Molte lingue sono dotate di `with` dichiarazione che consente ai programmatori di omettere il ricevitore di chiamate di metodo.

`with` può essere facilmente emulato in Ruby usando `instance_eval` :

```
def with(object, &block)
  object.instance_eval &block
end
```

Il metodo `with` può essere utilizzato per eseguire senza problemi metodi sugli oggetti:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

### Definizione dei metodi in modo dinamico

Con Ruby puoi modificare la struttura del programma in tempo di esecuzione. Un modo per farlo è definire i metodi in modo dinamico usando il metodo `method_missing` .

Diciamo che vogliamo essere in grado di verificare se un numero è maggiore di un altro numero con la sintassi `777.is_greater_than_123?` .

```
# open Numeric class
class Numeric
  # override `method_missing`
  def method_missing(method_name,*args)
```

```
    # test if the method_name matches the syntax we want
    if method_name.to_s.match /^is_greater_than_(\d+)\?$/
      # capture the number in the method_name
      the_other_number = $1.to_i
      # return whether the number is greater than the other number or not
      self > the_other_number
    else
      # if the method_name doesn't match what we want, let the previous definition of
`method_missing` handle it
      super
    end
  end
end
```

Una cosa importante da ricordare quando si usa `method_missing` anche quello di sovrascrivere `respond_to?` metodo:

```
class Numeric
   def respond_to?(method_name, include_all = false)
     method_name.to_s.match(/^is_greater_than_(\d+)\?$/) || super
   end
end
```

Dimenticare di farlo porta a una situazione incoerente, quando è possibile chiamare correttamente `600.is_greater_than_123` , ma `600.respond_to(:is_greater_than_123)` restituisce false.

## Definizione dei metodi su istanze

In ruby puoi aggiungere metodi a istanze esistenti di qualsiasi classe. Ciò consente di aggiungere un comportamento e un'istanza di una classe senza modificare il comportamento del resto delle istanze di quella classe.

```
class Example
  def method1(foo)
    puts foo
  end
end

#defines method2 on object exp
exp = Example.new
exp.define_method(:method2) {puts "Method2"}

#with method parameters
exp.define_method(:method3) {|name| puts name}
```

## metodo send ()

`send()` è usato per passare il messaggio `object` . `send()` è un metodo di istanza della classe `Object` . Il primo argomento di `send()` è il messaggio che stai inviando all'oggetto, cioè il nome di un metodo. Potrebbe essere una `string` o un `symbol` ma i **simboli** sono preferiti. Quindi gli argomenti che devono passare nel metodo, quelli saranno gli argomenti rimanenti in `send()` .

```
class Hello
```

```
  def hello(*args)
    puts 'Hello ' + args.join(' ')
  end
end
h = Hello.new
h.send :hello, 'gentle', 'readers'   #=> "Hello gentle readers"
# h.send(:hello, 'gentle', 'readers') #=> Here :hello is method and rest are the arguments to
method.
```

# Ecco l'esempio più descrittivo

```
class Account
  attr_accessor :name, :email, :notes, :address

  def assign_values(values)
    values.each_key do |k, v|
      # How send method would look a like
      # self.name = value[k]
      self.send("#{k}=", values[k])
    end
  end
end

user_info = {
  name: 'Matt',
  email: 'test@gms.com',
  address: '132 random st.',
  notes: "annoying customer"
}

account = Account.new
If attributes gets increase then we would messup the code
#--------- Bad way --------------
account.name = user_info[:name]
account.address = user_info[:address]
account.email = user_info[:email]
account.notes = user_info[:notes]

# --------- Meta Programing way --------------
account.assign_values(user_info) # With single line we can assign n number of attributes

puts account.inspect
```

Nota: `send()` stesso non è più raccomandato. Usa `__send__()` che ha il potere di chiamare metodi privati o (consigliato) `public_send()`

Leggi metaprogrammazione online: https://riptutorial.com/it/ruby/topic/5023/metaprogrammazione

# Capitolo 43: method_missing

## Parametri

| Parametro | Dettagli |
|---|---|
| metodo | Il nome del metodo che è stato chiamato (nell'esempio precedente si tratta di `:say_moo` , si noti che questo è un simbolo. |
| * args | Gli argomenti passati a questo metodo. Può essere qualsiasi numero o nessuno |
| &bloccare | Il blocco del metodo chiamato, può essere un blocco `do` o un blocco `{ }` chiuso |

## Osservazioni

Chiama sempre super, in fondo a questa funzione. Ciò consente di risparmiare un errore silenzioso quando viene chiamato qualcosa e non si riceve un errore.

Ad esempio, questo metodo_missing causerà problemi:

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say"
      speak
    end
  end
end

=> Animal.new.foobar
=> nil # This should really be raising an error
```

`method_missing` è un buon strumento da usare quando appropriato, ma ha due costi da considerare. In primo luogo, `method_missing` è meno efficiente: ruby deve cercare la classe e tutti i suoi antenati prima di poter `method_missing` a questo approccio; questa penalizzazione delle prestazioni può essere banale in un caso semplice, ma può sommarsi. Secondo e più ampiamente, questa è una forma di meta-programmazione che ha un grande potere che viene fornito con la responsabilità di assicurare che l'implementazione sia sicura, gestisce correttamente input dannosi, input inaspettati e così via.

Dovresti anche ignorare `respond_to_missing?` così:

```
class Animal
  def respond_to_missing?(method, include_private = false)
    method.to_s.start_with?("say_") || super
  end
end
```

```
=> Animal.new.respond_to?(:say_moo) # => true
```

# Examples

## Cattura di chiamate a un metodo indefinito

```
class Animal
  def method_missing(method, *args, &block)
    "Cannot call #{method} on Animal"
  end
end

=> Animal.new.say_moo
> "Cannot call say_moo on Animal"
```

## Usando il metodo mancante

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say"
      speak
    else
      super
    end
  end
end

=> a = Animal.new
=> a.say_moo
=> "moo"
=> a.shout_moo
=> NoMethodError: undefined method `shout_moo'
```

## Utilizzare con blocco

```
class Animal
  def method_missing(method, *args, &block)
    if method.to_s == 'say'
      block.call
    else
      super
    end
  end
end

 => Animal.new.say{ 'moo' }
 => "moo"
```

## Utilizzare con parametro

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say" && speak
      return speak.upcase if args.first == "shout"
      speak
    else
      super
    end
  end
end

=> Animal.new.say_moo
=> "moo"
=> Animal.new.say_moo("shout")
=> "MOO"
```

Leggi method_missing online: https://riptutorial.com/it/ruby/topic/1076/method-missing

# Capitolo 44: metodi

## introduzione

Le funzioni in Ruby forniscono un codice organizzato e riutilizzabile per preformare un insieme di azioni. Le funzioni semplificano il processo di codifica, impediscono la logica ridondante e rendono il codice più facile da seguire. Questo argomento descrive la dichiarazione e l'utilizzo di funzioni, argomenti, parametri, dichiarazioni di rendimento e ambito in Ruby.

## Osservazioni

Un **metodo** è un blocco di codice denominato, associato a uno o più oggetti e generalmente identificato da un elenco di parametri oltre al nome.

```
def hello(name)
  "Hello, #{name}"
end
```

Un richiamo di metodo specifica il nome del metodo, l'oggetto su cui deve essere richiamato (talvolta chiamato il ricevente) e zero o più valori dell'argomento che sono assegnati ai parametri del metodo named. Il valore dell'ultima espressione valutata nel metodo diventa il valore dell'espressione di chiamata del metodo.

```
hello("World")
# => "Hello, World"
```

Quando il ricevitore non è esplicito, è un `self` .

```
self
# => main

self.hello("World")
# => "Hello, World"
```

Come spiegato nel libro *Ruby Programming Language* , molte lingue distinguono tra funzioni, che non hanno oggetti associati e metodi, che sono invocati su un oggetto ricevente. Poiché Ruby è un linguaggio puramente orientato agli oggetti, tutti i metodi sono metodi veri e sono associati ad almeno un oggetto.

## Panoramica dei parametri del metodo

| genere | Metodo Signature | Esempio di chiamata | assegnazioni |
|---|---|---|---|
| **R** equo | `def fn(a,b,c)` | `fn(2,3,5)` | `a=2, b=3, c=5` |
| **V** ariadic | `def fn(*rest)` | `fn(2,3,5)` | `rest=[2, 3, 5]` |

| genere | Metodo Signature | Esempio di chiamata | assegnazioni |
|---|---|---|---|
| **D** efault | `def fn(a=0,b=1)` | `fn(2,3)` | `a=2, b=3` |
| **K** eyword | `def fn(a:0,b:1)` | `fn(a:2,b:3)` | `a=2, b=3` |

Questi tipi di argomenti possono essere combinati praticamente in ogni modo possibile per creare funzioni variadiche. Il numero minimo di argomenti per la funzione sarà uguale alla quantità di argomenti richiesti nella firma. Gli argomenti extra verranno assegnati prima ai parametri predefiniti, quindi al parametro `*rest` .

| genere | Metodo Signature | Esempio di chiamata | assegnazioni |
|---|---|---|---|
| R, S, V, R | `def fn(a,b=1,*mid,z)` | `fn(2,97)` | `a=2, b=1, mid=[], z=97` |
| | | `fn(2,3,97)` | `a=2, b=3, mid=[], z=97` |
| | | `fn(2,3,5,97)` | `a=2, b=3, mid=[5], z=97` |
| | | `fn(2,3,5,7,97)` | `a=2, b=3, mid=[5,7], z=97` |
| R, K, K | `def fn(a,g:6,h:7)` | `fn(2)` | `a=2, g=6, h=7` |
| | | `fn(2,h:19)` | `a=2, g=6, h=19` |
| | | `fn(2,g:17,h:19)` | `a=2, g=17, h=19` |
| VK | `def fn(**ks)` | `fn(a:2,g:17,h:19)` | `ks={a:2, g:17, h:19}` |
| | | `fn(four:4,five:5)` | `ks={four:4, five:5}` |

# Examples

## Singolo parametro richiesto

```
def say_hello_to(name)
    puts "Hello #{name}"
end
```

---

```
say_hello_to('Charles')    # Hello Charles
```

## Più parametri richiesti

```
def greet(greeting, name)
    puts "#{greeting} #{name}"
end
```

---

```
greet('Hi', 'Sophie')    # Hi Sophie
```

## Parametri di default

```
def make_animal_sound(sound = 'Cuack')
    puts sound
end
```

---

```
make_animal_sound('Mooo') # Mooo
make_animal_sound         # Cuack
```

È possibile includere i valori predefiniti per più argomenti:

```
def make_animal_sound(sound = 'Cuack', volume = 11)
    play_sound(sound, volume)
end

make_animal_sound('Mooo') # Spinal Tap cow
```

Tuttavia, non è possibile fornire il secondo senza fornire anche il primo. Invece di utilizzare i parametri posizionali, prova i parametri delle parole chiave:

```
def make_animal_sound(sound: 'Cuack', volume: 11)
    play_sound(sound, volume)
end

make_animal_sound(volume: 1) # Duck whisper
```

O un parametro hash che memorizza le opzioni:

```
def make_animal_sound(options = {})
    options[:sound]  ||= 'Cuak'
    options[:volume] ||= 11
    play_sound(sound, volume)
end

make_animal_sound(:sound => 'Mooo')
```

I valori dei parametri predefiniti possono essere impostati da qualsiasi espressione di ruby. L'espressione verrà eseguita nel contesto del metodo, quindi puoi anche dichiarare le variabili locali qui. Nota, non passerà attraverso la revisione del codice. Per gentile concessione di caius per averlo indicato .

```
def make_animal_sound( sound = ( raise 'TUU-too-TUU-too...' ) ); p sound; end

make_animal_sound 'blaaaa' # => 'blaaaa'
make_animal_sound          # => TUU-too-TUU-too... (RuntimeError)
```

## Parametro opzionale (operatore splat)

```
def welcome_guests(*guests)
    guests.each { |guest| puts "Welcome #{guest}!" }
end
```

---

```
welcome_guests('Tom')    # Welcome Tom!
welcome_guests('Rob', 'Sally', 'Lucas') # Welcome Rob!
                                        # Welcome Sally!
                                        # Welcome Lucas!
```

Nota che `welcome_guests(['Rob', 'Sally', 'Lucas'])` mostrerà `Welcome ["Rob", "Sally", "Lucas"]!` Invece, se hai una lista, puoi fare `welcome_guests(*['Rob', 'Sally', 'Lucas'])` e funzionerà come `welcome_guests('Rob', 'Sally', 'Lucas')` .

## Mix di parametri facoltativo predefinito richiesto

```
def my_mix(name,valid=true, *opt)
    puts name
    puts valid
    puts opt
end
```

Chiama come segue:

```
my_mix('me')
# 'me'
# true
# []

my_mix('me', false)
# 'me'
# false
# []

my_mix('me', true, 5, 7)
# 'me'
# true
# [5,7]
```

## Le definizioni di metodo sono espressioni

La definizione di un metodo in Ruby 2.x restituisce un simbolo che rappresenta il nome:

```
class Example
  puts def hello
  end
end

#=> :hello
```

Ciò consente interessanti tecniche di metaprogrammazione. Ad esempio, i metodi possono essere impacchettati con altri metodi:

```
class Class
  def logged(name)
    original_method = instance_method(name)
    define_method(name) do |*args|
      puts "Calling #{name} with #{args.inspect}."
      original_method.bind(self).call(*args)
      puts "Completed #{name}."
    end
  end
end

class Meal
  def initialize
    @food = []
  end

  logged def add(item)
    @food << item
  end
end

meal = Meal.new
meal.add "Coffee"
# Calling add with ["Coffee"].
# Completed add.
```

## Catturare argomenti di parole chiave non dichiarate (double splat)

L'operatore `**` funziona in modo simile all'operatore `*` ma si applica ai parametri delle parole chiave.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

options(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> { :foo => "Foo!", :bar => "Bar!" }
```

Nell'esempio precedente, se non si utilizza `**other_options` viene `**other_options` un `ArgumentError: unknown keyword: foo, bar` verrebbe generato `ArgumentError: unknown keyword: foo, bar` errore nella `ArgumentError: unknown keyword: foo, bar` .

```
def without_double_splat(required_key:, optional_key: nil)
  # do nothing
end

without_double_splat(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> ArgumentError: unknown keywords: foo, bar
```

Questo è utile quando hai un hash di opzioni che vuoi passare ad un metodo e non vuoi filtrare le chiavi.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end
```

```
my_hash = { required_key: true, foo: 'Foo!', bar: 'Bar!' }

options(my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

È anche possibile *decomprimere* un hash utilizzando l'operatore `**` . Ciò consente di fornire parole chiave direttamente a un metodo oltre ai valori di altri hash:

```
my_hash = { foo: 'Foo!', bar: 'Bar!' }

options(required_key: true, **my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

## Cedendo ai blocchi

Puoi inviare un blocco al tuo metodo e può chiamare quel blocco più volte. Questo può essere fatto inviando un proc / lambda o tale, ma è più facile e veloce con `yield` :

```
def simple(arg1,arg2)
  puts "First we are here:  #{arg1}"
  yield
  puts "Finally we are here:  #{arg2}"
  yield
end
simple('start','end') { puts "Now we are inside the yield" }

#> First we are here:  start
#> Now we are inside the yield
#> Finally we are here:  end
#> Now we are inside the yield
```

Nota che `{ puts ... }` non è all'interno delle parentesi, viene implicitamente dopo. Questo significa anche che possiamo avere solo un blocco di `yield` . Possiamo passare argomenti alla `yield` :

```
def simple(arg)
  puts "Before yield"
  yield(arg)
  puts "After yield"
end
simple('Dave') { |name| puts "My name is #{name}" }

#> Before yield
#> My name is Dave
#> After yield
```

Con yield possiamo facilmente fare iteratori o funzioni che funzionano su altro codice:

```
def countdown(num)
  num.times do |i|
    yield(num-i)
  end
end
```

```
countdown(5) { |i| puts "Call number #{i}" }
```

```
#> Call number 5
#> Call number 4
#> Call number 3
#> Call number 2
#> Call number 1
```

In effetti, è con il `yield` che cose come `foreach` , `each` e le `times` sono generalmente implementate nelle classi.

Se vuoi scoprire se ti è stato dato un blocco o no, usa `block_given?` :

```
class Employees
  def names
    ret = []
    @employees.each do |emp|
      if block_given?
        yield(emp.name)
      else
        ret.push(emp.name)
      end
    end

    ret
  end
end
```

In questo esempio si presuppone che la classe `Employees` abbia un elenco `@employees` che può essere iterato con `each` per ottenere oggetti che hanno nomi di dipendenti che utilizzano il metodo `name` . Se ci viene dato un blocco, poi ci `yield` il nome al blocco, altrimenti basta spingerlo a una matrice che ci ritorno.

## Tuple Arguments

Un metodo può prendere un parametro di matrice e destrutturarlo immediatamente in variabili locali con nome. Trovato sul blog di Mathias Meyer .

```
def feed( amount, (animal, food) )

    p "#{amount} #{animal}s chew some #{food}"

end

feed 3, [ 'rabbit', 'grass' ] # => "3 rabbits chew some grass"
```

## Definire un metodo

I metodi sono definiti con la parola chiave `def` , seguita dal *nome* del *metodo* e da un elenco opzionale di *nomi parametro* tra parentesi. Il codice Ruby tra `def` e `end` rappresenta il *corpo* del metodo.

```
def hello(name)
  "Hello, #{name}"
end
```

Un richiamo di metodo specifica il nome del metodo, l'oggetto su cui deve essere richiamato (talvolta chiamato il ricevente) e zero o più valori dell'argomento che sono assegnati ai parametri del metodo named.

```
hello("World")
# => "Hello, World"
```

Quando il ricevitore non è esplicito, è un `self` .

I nomi dei parametri possono essere utilizzati come variabili all'interno del corpo del metodo e i valori di questi parametri denominati derivano dagli argomenti in una chiamata di metodo.

```
hello("World")
# => "Hello, World"
hello("All")
# => "Hello, All"
```

## Utilizzare una funzione come blocco

Molte funzioni in Ruby accettano un blocco come argomento. Per esempio:

```
[0, 1, 2].map {|i| i + 1}
 => [1, 2, 3]
```

Se hai già una funzione che fa quello che vuoi, puoi trasformarlo in un blocco usando `&method(:fn)` :

```
def inc(num)
   num + 1
end

[0, 1, 2].map &method(:inc)
 => [1, 2, 3]
```

Leggi metodi online: https://riptutorial.com/it/ruby/topic/997/metodi

# Capitolo 45: Modificatori di accesso rubino

## introduzione

Controllo di accesso (scope) a vari metodi, membri di dati, metodi di inizializzazione.

## Examples

### Variabili di istanza e variabili di classe

Iniziamo a rispolverare le **variabili di istanza:** si comportano più come proprietà di un oggetto. Sono inizializzati su una creazione di un oggetto. Le variabili di istanza sono accessibili tramite i metodi di istanza. Per oggetto ha variabili di istanza. Le variabili d'istanza non sono condivise tra oggetti.

La classe di sequenze ha @from, @to e @by come variabili di istanza.

```
class Sequence
    include Enumerable

    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
5
7
```

```
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

Output:
1
4
7
```

**Variabili di** classe Tratta le variabili di classe come variabili statiche di java, che sono condivise tra i vari oggetti di quella classe. Le variabili di classe sono archiviate nella memoria heap.

```
class Sequence
    include Enumerable
    @@count = 0
    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
        @@count = @@count + 1
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end

    def getCount
        @@count
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
5
7
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
```

```
    puts x
end

Output:
1
4
7

puts object1.getCount
Output: 2
```

Condiviso tra oggetto e oggetto1.

**Confronto tra le istanze e le variabili di classe di Ruby contro Java:**

```
Class Sequence{
    int from, to, by;
    Sequence(from, to, by){// constructor method of Java is equivalent to initialize method of
ruby
        this.from = from;// this.from of java is equivalent to @from indicating
currentObject.from
        this.to = to;
        this.by = by;
    }
    public void each(){
        int x = this.from;//objects attributes are accessible in the context of the object.
        while x > this.to
            x = x + this.by
    }
}
```

## Controlli di accesso

**Confronto dei controlli di accesso di Java contro Ruby:** se il metodo è dichiarato privato in Java, è possibile accedervi solo tramite altri metodi all'interno della stessa classe. Se un metodo è dichiarato protetto, è possibile accedervi da altre classi presenti nello stesso pacchetto e sottoclassi della classe in un pacchetto diverso. Quando un metodo è pubblico è visibile a tutti. In Java, il concetto di visibilità del controllo di accesso dipende da dove queste classi si trovano nella gerarchia ereditari / dei pacchetti.

**Mentre in Ruby, la gerarchia dell'ereditarietà o il pacchetto / modulo non si adattano. È tutto su quale oggetto è il destinatario di un metodo** .

**Per un metodo privato in Ruby** , non può mai essere chiamato con un ricevitore esplicito. Possiamo (solo) chiamare il metodo privato con un ricevitore implicito.

Ciò significa anche che possiamo chiamare un metodo privato all'interno di una classe dichiarata e di tutte le sottoclassi di questa classe.

```
class Test1
  def main_method
    method_private
  end
```

```
  private
  def method_private
    puts "Inside methodPrivate for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_private
  end
end

Test1.new.main_method
Test2.new.main_method

Inside methodPrivate for Test1
Inside methodPrivate for Test2

class Test3 < Test1
  def main_method
    self.method_private #We were trying to call a private method with an explicit receiver and
if called in the same class with self would fail.
  end
end

Test1.new.main_method
This will throw NoMethodError

You can never call the private method from outside the class hierarchy where it was defined.
```

**Il metodo protetto** può essere chiamato con un ricevitore implicito, come se fosse privato. Inoltre il metodo protetto può anche essere chiamato da un ricevitore esplicito (solo) se il ricevitore è "self" o "un oggetto della stessa classe".

```
class Test1
  def main_method
    method_protected
  end

  protected
  def method_protected
    puts "InSide method_protected for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_protected # called by implicit receiver
  end
end

class Test3 < Test1
  def main_method
    self.method_protected # called by explicit receiver "an object of the same class"
  end
end


InSide method_protected for Test1
```

```
InSide method_protected for Test2
InSide method_protected for Test3


class Test4 < Test1
  def main_method
    Test2.new.method_protected # "Test2.new is the same type of object as self"
  end
end

Test4.new.main_method

class Test5
  def main_method
    Test2.new.method_protected
  end
end

Test5.new.main_method
This would fail as object Test5 is not subclass of Test1
```

**Considera i metodi pubblici con la massima visibilità**

**Sommario**

1. **Pubblico:** i metodi pubblici hanno la massima visibilità
2. **Protetto: il metodo protetto** può essere chiamato con un ricevitore implicito, come se fosse privato. Inoltre il metodo protetto può anche essere chiamato da un ricevitore esplicito (solo) se il ricevitore è "self" o "un oggetto della stessa classe".
3. **Privato: per un metodo privato in Ruby** , non può mai essere chiamato con un ricevitore esplicito. Possiamo (solo) chiamare il metodo privato con un ricevitore implicito. Ciò significa anche che possiamo chiamare un metodo privato all'interno di una classe dichiarata e di tutte le sottoclassi di questa classe.

Leggi Modificatori di accesso rubino online: https://riptutorial.com/it/ruby/topic/10797/modificatori-di-accesso-rubino

# Capitolo 46: moduli

## Sintassi

- Dichiarazione

```
module Name;

    any ruby expressions;

end
```

## Osservazioni

I nomi dei moduli in Ruby sono costanti, quindi devono iniziare con una lettera maiuscola.

```
module foo; end # Syntax error: class/module name must be CONSTANT
```

## Examples

### Un semplice mixin con include

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  include SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz          # => "baz!"
b.foo          # => "foo!"
# works thanks to the mixin
```

Now `Bar` è un mix dei suoi metodi e dei metodi di `SomeMixin` .

Si noti che il modo in cui un mixin viene utilizzato in una classe dipende da come viene aggiunto:

- la parola chiave `include` valuta il codice del modulo nel contesto della classe (ad esempio, le definizioni del metodo saranno metodi sulle istanze della classe),
- `extend` il codice del modulo nel contesto della classe singleton dell'oggetto (i metodi sono disponibili direttamente sull'oggetto esteso).

## Modulo come spazio dei nomi

I moduli possono contenere altri moduli e classi:

```
module Namespace

    module Child

        class Foo; end

    end # module Child

    # Foo can now be accessed as:
    #
    Child::Foo

end # module Namespace

# Foo must now be accessed as:
#
Namespace::Child::Foo
```

## Un semplice mixin con estensione

Un mixin è solo un modulo che può essere aggiunto (mescolato in) a una classe. un modo per farlo è con il metodo estendere. Il metodo di `extend` aggiunge metodi del mixin come metodi di classe.

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  extend SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz         # => "baz!"
b.foo         # NoMethodError, as the method was NOT added to the instance
Bar.foo       # => "foo!"
# works only on the class itself
```

## Moduli e composizione di classe

È possibile utilizzare i moduli per creare classi più complesse attraverso la *composizione* . La direttiva `include ModuleName` incorpora i metodi di un modulo in una classe.

```
module Foo
  def foo_method
```

```
    puts 'foo_method called!'
  end
end

module Bar
  def bar_method
    puts 'bar_method called!'
  end
end

class Baz
  include Foo
  include Bar

  def baz_method
    puts 'baz_method called!'
  end
end
```

`Baz` ora contiene metodi di `Foo` e `Bar` oltre ai propri metodi.

```
new_baz = Baz.new
new_baz.baz_method #=> 'baz_method called!'
new_baz.bar_method #=> 'bar_method called!'
new_baz.foo_method #=> 'foo_method called!'
```

Leggi moduli online: https://riptutorial.com/it/ruby/topic/4039/moduli

# Capitolo 47: Monkey Patching in Ruby

## introduzione

Monkey Patching è un modo per modificare ed estendere le classi in Ruby. In sostanza, è possibile modificare le classi già definite in Ruby, aggiungendo nuovi metodi e persino modificando i metodi precedentemente definiti.

## Osservazioni

La patch delle scimmie viene spesso utilizzata per modificare il comportamento del codice rubino esistente, ad esempio dalle gemme.

Per esempio, vedi questo succo .

Può anche essere usato per estendere classi ruby esistenti come Rails fa con ActiveSupport, eccone un esempio .

## Examples

### Cambiare qualsiasi metodo

```
def hello
  puts "Hello readers"
end

hello # => "Hello readers"

def hello
  puts "Hell riders"
end

hello # => "Hell riders"
```

### Modifica di un metodo esistente in ruby

```
puts "Hello readers".reverse # => "sredaer olleH"

class String
  def reverse
    "Hell riders"
  end
end

puts "Hello readers".reverse # => "Hell riders"
```

### Modifica di un metodo con parametri

È possibile accedere allo stesso contesto esatto del metodo che si sovrascrive.

```
class Boat
  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

puts Boat.new("Doat").name # => "Doat"

class Boat
  def name
    "□ #{@name} □"
  end
end

puts Boat.new("Moat").name # => "□ Moat □"
```

## Estendere una classe esistente

```
class String
  def fancy
    "~~~{#{self}}~~~"
  end
end

puts "Dorian".fancy # => "~~~{Dorian}~~~"
```

## Scimmia sicura che si aggiusta con i perfezionamenti

Dal momento che Ruby 2.0, Ruby consente di avere Patch Patching più sicuro con perfezionamenti. Fondamentalmente consente di limitare il codice Monkey Patched per applicare solo quando richiesto.

Per prima cosa creiamo un perfezionamento in un modulo:

```
module RefiningString
  refine String do
    def reverse
      "Hell riders"
    end
  end
end
```

Quindi possiamo decidere dove usarlo:

```
class AClassWithoutMP
  def initialize(str)
    @str = str
  end
```

```
  def reverse
    @str.reverse
  end
end

class AClassWithMP
  using RefiningString

  def initialize(str)
    @str = str
  end

  def reverse
    str.reverse
  end
end

AClassWithoutMP.new("hello".reverse # => "olle"
AClassWithMP.new("hello").reverse # "Hell riders"
```

Leggi Monkey Patching in Ruby online: https://riptutorial.com/it/ruby/topic/6043/monkey-patching-in-ruby

# Capitolo 48: Monkey Patching in Ruby

## Examples

### Scimmia che rattoppa una classe

La patch delle scimmie è la modifica di classi o oggetti al di fuori della classe stessa.

A volte è utile aggiungere funzionalità personalizzate.

**Esempio:** sovrascrivere la classe stringa per fornire l'analisi su booleano

```
class String
  def to_b
    self =~ (/^(true|TRUE|True|1)$/i) ? true : false
  end
end
```

Come puoi vedere, aggiungiamo il metodo `to_b()` alla classe String, quindi possiamo analizzare qualsiasi stringa con un valore booleano.

```
>>'true'.to_b
=> true
>>'foo bar'.to_b
=> false
```

### Scimmia che rattoppa un oggetto

Come il patching delle classi, puoi anche applicare patch a singoli oggetti. La differenza è che solo quell'istanza può utilizzare il nuovo metodo.

Esempio: sovrascrivere un oggetto stringa per fornire l'analisi su booleano

```
s = 'true'
t = 'false'

def s.to_b
  self =~ /true/ ? true : false
end

>> s.to_b
=> true
>> t.to_b
=> undefined method `to_b' for "false":String (NoMethodError)
```

Leggi Monkey Patching in Ruby online: https://riptutorial.com/it/ruby/topic/6228/monkey-patching-in-ruby

# Capitolo 49: Monkey Patching in Ruby

## Osservazioni

Le patch per le scimmie, sebbene convenienti, presentano alcune insidie che non sono immediatamente evidenti. In particolare, una patch come quella nell'esempio inquina l'ambito globale. Se due moduli aggiungono entrambi il `Hash#symbolize` , solo l'ultimo modulo richiesto applica effettivamente la sua modifica; il resto viene cancellato.

Inoltre, se c'è un errore in un metodo patchato, lo stacktrace punta semplicemente alla classe patchata. Ciò implica che c'è un bug nella classe `Hash` stessa (che esiste ora).

Infine, poiché Ruby è molto flessibile con i contenitori da tenere, un metodo che sembra molto semplice quando lo scrivi ha molte funzionalità indefinite. Ad esempio, la creazione della `Array#sum` è utile per una matrice di numeri, ma si interrompe quando viene fornita una matrice di una classe personalizzata.

Un'alternativa più sicura è la raffinatezza, disponibile in Ruby> = 2.0.

## Examples

### Aggiunta di funzionalità

Puoi aggiungere un metodo a qualsiasi classe in Ruby, indipendentemente dal fatto che sia un built-in o meno. L'oggetto chiamante viene referenziato usando `self` .

```
class Fixnum
  def plus_one
    self + 1
  end

  def plus(num)
    self + num
  end

  def concat_one
    self.to_s + '1'
  end
end

1.plus_one # => 2
3.plus(5) # => 8
6.concat_one # => '61'
```

Leggi Monkey Patching in Ruby online: https://riptutorial.com/it/ruby/topic/6616/monkey-patching-in-ruby

# Capitolo 50: Numeri

## Osservazioni

## Gerarchia dei numeri

Ruby include diverse classi built-in per rappresentare i numeri:

```
Numeric
  Integer
    Fixnum     # 1
    Bignum     # 10000000000000000000
  Float        # 1.0
  Complex      # (1+0i)
  Rational     # Rational(2, 3) == 2/3
  BigDecimal   # not loaded by default
```

I più comuni sono:

- `Fixnum` da rappresentare, ad esempio interi positivi e negativi
- `Float` per rappresentare numeri in virgola mobile

`BigDecimal` è l'unico non caricato per impostazione predefinita. Puoi caricarlo con:

```
require "bigdecimal"
```

Si noti che in Ruby 2.4+, `Fixnum` e `Bignum` sono unificati; *tutti gli* interi ora sono solo membri della classe `Integer` . Per compatibilità all'indietro, `Fixnum == Bignum == Integer` .

## Examples

### Creare un intero

```
0       # creates the Fixnum 0
123     # creates the Fixnum 123
1_000   # creates the Fixnum 1000. You can use _ as separator for readability
```

Di default la notazione è base 10. Tuttavia, ci sono altre notazioni incorporate per basi differenti:

```
0xFF    # Hexadecimal representation of 255, starts with a 0x
0b100   # Binary representation of 4, starts with a 0b
0555    # Octal representation of 365, starts with a 0 and digits
```

### Convertire una stringa in numero intero

È possibile utilizzare il metodo `Integer` per convertire una `String` in un `Integer` :

```
Integer("123")      # => 123
Integer("0xFF")     # => 255
Integer("0b100")    # => 4
Integer("0555")     # => 365
```

È inoltre possibile passare un parametro di base al metodo `Integer` per convertire i numeri da una determinata base

```
Integer('10', 5)    # => 5
Integer('74', 8)    # => 60
Integer('NUM', 36)  # => 30910
```

Si noti che il metodo genera un `ArgumentError` se il parametro non può essere convertito:

```
Integer("hello")
# raises ArgumentError: invalid value for Integer(): "hello"
Integer("23-hello")
# raises ArgumentError: invalid value for Integer(): "23-hello"
```

Puoi anche utilizzare il metodo `String#to_i` . Tuttavia, questo metodo è leggermente più permissivo e ha un comportamento diverso da `Integer` :

```
"23".to_i         # => 23
"23-hello".to_i   # => 23
"hello".to_i      # => 0
```

`String#to_i` accetta un argomento, la base per interpretare il numero come:

```
"10".to_i(2) # => 2
"10".to_i(3) # => 3
"A".to_i(16) # => 10
```

## Convertire un numero in una stringa

Fixnum # to_s accetta un argomento di base opzionale e rappresenta il numero specificato in quella base:

```
2.to_s(2)   # => "10"
3.to_s(2)   # => "11"
3.to_s(3)   # => "10"
10.to_s(16) # => "a"
```

Se non viene fornito alcun argomento, allora rappresenta il numero in base 10

```
2.to_s # => "2"
10423.to_s # => "10423"
```

## Dividere due numeri

Quando dividete due numeri fate attenzione al tipo che volete in cambio. Si noti che la divisione di

**due numeri interi invocherà la divisione intera** . Se il tuo obiettivo è eseguire la divisione float, almeno uno dei parametri dovrebbe essere di tipo `float` .

Divisione intera:

```
3 / 2 # => 1
```

Divisione Float

```
3 / 3.0 # => 1.0

16 / 2 / 2    # => 4
16 / 2 / 2.0  # => 4.0
16 / 2.0 / 2  # => 4.0
16.0 / 2 / 2  # => 4.0
```

## Numeri razionali

`Rational` rappresenta un numero razionale come numeratore e denominatore:

```
r1 = Rational(2, 3)
r2 = 2.5.to_r
r3 = r1 + r2
r3.numerator   # => 19
r3.denominator # => 6
Rational(2, 4) # => (1/2)
```

Altri modi di creare un Razionale

```
Rational('2/3')  # => (2/3)
Rational(3)      # => (3/1)
Rational(3, -5)  # => (-3/5)
Rational(0.2)    # => (3602879701896397/18014398509481984)
Rational('0.2')  # => (1/5)
0.2.to_r         # => (3602879701896397/18014398509481984)
0.2.rationalize  # => (1/5)
'1/4'.to_r       # => (1/4)
```

## Numeri complessi

```
1i     # => (0+1i)
1.to_c # => (1+0i)
rectangular = Complex(2, 3)  # => (2+3i)
polar       = Complex('1@2') # => (-0.4161468365471424+0.9092974268256817i)

polar.rectangular # => [-0.4161468365471424, 0.9092974268256817]
rectangular.polar # => [3.605551275463989, 0.982793723247329]
rectangular + polar # => (1.5838531634528576+3.909297426825682i)
```

## Numeri pari e dispari

Il `even?` il metodo può essere usato per determinare se un numero è pari

```
4.even?      # => true
5.even?      # => false
```

Lo `odd?` metodo può essere utilizzato per determinare se un numero è dispari

```
4.odd?       # => false
5.odd?       # => true
```

## Numeri arrotondati

Il metodo `round` arrotonda un numero se la prima cifra dopo la sua cifra decimale è 5 o superiore e arrotondata per difetto se tale cifra è 4 o inferiore. Questo contiene un argomento facoltativo per la precisione che stai cercando.

```
4.89.round        # => 5
4.25.round        # => 4
3.141526.round(1) # => 3.1
3.141526.round(2) # => 3.14
3.141526.round(4) # => 3.1415
```

I numeri in virgola mobile possono anche essere arrotondati al numero intero più alto rispetto al numero con il metodo `floor`

```
4.9999999999999.floor # => 4
```

Possono anche essere arrotondati al numero più basso più alto del numero usando il metodo `ceil`

```
4.0000000000001.ceil  # => 5
```

Leggi Numeri online: https://riptutorial.com/it/ruby/topic/1083/numeri

# Capitolo 51: Operatore Splat (*)

## Examples

### Array coercing nell'elenco dei parametri

Supponi di avere un array:

```
pair = ['Jack','Jill']
```

E un metodo che prende due argomenti:

```
def print_pair (a, b)
  puts "#{a} and #{b} are a good couple!"
end
```

Potresti pensare di poter passare semplicemente l'array:

```
print_pair(pair) # wrong number of arguments (1 for 2) (ArgumentError)
```

Dato che l'array è solo un argomento, non due, quindi Ruby lancia un'eccezione. Si *potrebbe* tirare fuori ogni elemento singolarmente:

```
print_pair(pair[0], pair[1])
```

Oppure puoi usare l'operatore splat per risparmiarti qualche sforzo:

```
print_pair(*pair)
```

### Numero variabile di argomenti

L'operatore di splat rimuove i singoli elementi di un array e li trasforma in un elenco. Questo è più comunemente usato per creare un metodo che accetta un numero variabile di argomenti:

```
# First parameter is the subject and the following parameters are their spouses
def print_spouses(person, *spouses)
  spouses.each do |spouse|
    puts "#{person} married #{spouse}."
  end
end

print_spouses('Elizabeth', 'Conrad', 'Michael', 'Mike', 'Eddie', 'Richard', 'John', 'Larry')
```

Si noti che un array conta solo come un elemento nell'elenco, quindi sarà necessario anche l'operatore splat sul lato chiamante se si dispone di un array che si desidera passare:

```
bonaparte = ['Napoleon','Joséphine','Marie Louise']
print_spouses(*bonaparte)
```

Leggi Operatore Splat (*) online: https://riptutorial.com/it/ruby/topic/9862/operatore-splat----

# Capitolo 52: operatori

## Osservazioni

### Gli operatori sono metodi

La maggior parte degli operatori sono in realtà solo metodi, quindi `x + y` sta chiamando il metodo `+` di `x` con l'argomento `y` , che dovrebbe essere scritto `x.+(y)` . Se scrivi un tuo metodo che ha il significato semantico di un determinato operatore, puoi implementare la tua variante nella classe.

Come un esempio sciocco:

```
# A class that lets you operate on numbers by name.
class NamedInteger
  name_to_value = { 'one' => 1, 'two' => 2, ... }

  # define the plus method
  def + (left_addend, right_addend)
    name_to_value(left_addend) + name_to_value(right_addend)
  end

  ...
end
```

### Quando usare `&&` vs. `and` , `||` contro `or`

Nota che ci sono due modi per esprimere i booleani, sia `&&` o `and` , che `||` oppure `or` - sono spesso intercambiabili, ma non sempre. Ci riferiremo a queste come varianti di "carattere" e "parola".

Le varianti dei caratteri hanno una *precedenza* più alta, quindi riducono la necessità di parentesi in istruzioni più complesse per evitare errori imprevisti.

Le varianti di parole erano originariamente intese come *operatori di flusso di controllo* piuttosto che operatori booleani. Cioè, sono stati progettati per essere utilizzati in dichiarazioni di metodo concatenate:

```
raise 'an error' and return
```

Mentre *possono* essere usati come operatori booleani, la loro precedenza inferiore li rende imprevedibili.

In secondo luogo, molti rubyisti preferiscono la variante del carattere quando creano un'espressione booleana (una che `x.nil? || x.empty?` `true` o `false` ) come `x.nil? || x.empty?` . D'altra parte, le varianti di parole sono preferite nei casi in cui una *serie di metodi* sono in fase di valutazione e uno può fallire. Ad esempio, un idioma comune che utilizza la variante di parola per metodi che restituiscono `nil` in caso di errore potrebbe essere simile a:

```
def deliver_email
```

```
  # If the first fails, try the backup, and if that works, all good
  deliver_by_primary or deliver_by_backup and return
  # error handling code
end
```

# Examples

## Precedenza e metodi dell'operatore

Dal più alto al più basso, questa è la tabella di precedenza per Ruby. Le operazioni con precedenza elevata avvengono prima delle operazioni a bassa precedenza.

| Operators | Operations | Method? |
|---|---|---|
| . | Method call (e.g. foo.bar) | |
| [] []= | Bracket Lookup, Bracket Set | ✓[1] |
| ! ~ + | Boolean NOT, complement, unary plus | ✓[2] |
| ** | Exponentiation | ✓ |
| - | Unary minus | ✓[2] |
| * / % | Multiplication, division, modulo | ✓ |
| + - | Addition, subtraction | ✓ |
| << >> | Bitwise shift | ✓ |
| & | Bitwise AND | ✓ |
| \| ^ | Bitwise OR, Bitwise XOR | ✓ |
| < <= >= > | Comparison | ✓ |
| <=> == != === =~ !~ | Equality, pattern matching, comparison | ✓[3] |
| && | Boolean AND | |
| \|\| | Boolean OR | |
| .. ... | Inclusive range, Exclusive range | |
| ? : | Ternary operator | |
| rescue | Modifier rescue | |
| = += -= | Assignments | |
| defined? | Defined operator | |
| not | Boolean NOT | |
| or and | Boolean OR, Boolean AND | |
| if unless while until | Modifier if, unless, while, until | |
| { } | Block with braces | |
| do end | Block with do end | |

Unario + e unario - sono per `+obj` , `-obj` o `-(some_expression)` .

Modificatore-se, modificatore-a meno, ecc. Sono per le versioni modificatore di quelle parole chiave. Ad esempio, questa è una modifica, a meno di un'espressione:

```
a += 1 unless a.zero?
```

Gli operatori con ✓ possono essere definiti come metodi. La maggior parte dei metodi sono denominati esattamente come viene chiamato l'operatore, ad esempio:

```
class Foo
  def **(x)
```

```
    puts "Raising to the power of #{x}"
  end
  def <<(y)
    puts "Shifting left by #{y}"
  end
  def !
    puts "Boolean negation"
  end
end

Foo.new ** 2     #=> "Raising to the power of 2"
Foo.new << 3     #=> "Shifting left by 3"
!Foo.new         #=> "Boolean negation"
```

[1] I metodi Bracket Lookup e Bracket Set ( `[]` e `[]=` ) hanno i loro argomenti definiti dopo il nome, ad esempio:

```
class Foo
  def [](x)
    puts "Looking up item #{x}"
  end
  def []=(x,y)
    puts "Setting item #{x} to #{y}"
  end
end

f = Foo.new
f[:cats] = 42    #=> "Setting item cats to 42"
f[17]            #=> "Looking up item 17"
```

[2] Gli operatori "unary plus" e "unary-minus" sono definiti come metodi denominati `+@` e `-@` , ad esempio

```
class Foo
  def -@
    puts "unary minus"
  end
  def +@
    puts "unary plus"
  end
end

f = Foo.new
+f               #=> "unary plus"
-f               #=> "unary minus"
```

[3] Nelle prime versioni di Ruby l'operatore di disuguaglianza `!=` E l'operatore non corrispondente `!~` Non può essere definito come metodo. Invece, è stato invocato il metodo per l'operatore di uguaglianza corrispondente `==` o l'operatore di corrispondenza `=~` , e il risultato di tale metodo era booleano invertito da Ruby.

Se non definisci i tuoi operatori `!=` O `!~` comportamento sopra riportato è ancora vero. Tuttavia, a partire da Ruby 1.9.1, questi due operatori possono anche essere definiti come metodi:

```
class Foo
```

```
  def ==(x)
    puts "checking for EQUALITY with #{x}, returning false"
    false
  end
end

f = Foo.new
x = (f == 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "false"
x = (f != 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "true"

class Foo
  def !=(x)
    puts "Checking for INequality with #{x}"
  end
end

f != 42           #=> "checking for INequality with 42"
```

## Operatore di uguaglianza dei casi (===)

Conosciuto anche come *triple equals* .

Questo operatore non verifica l'uguaglianza, ma verifica se l'operando di destra ha una relazione IS A con l'operando di sinistra. In quanto tale, il famoso *operatore di uguaglianza dei casi* nome è fuorviante.

Questo SO risposta descrive così: il modo migliore per descrivere `a === b` è "se ho un cassetto etichettato `a` , ha senso mettere `b` in esso?" In altre parole, l'insieme `a` include il membro `b` ?

**Esempi ( fonte )**

```
(1..5) === 3            # => true
(1..5) === 6            # => false

Integer === 42          # => true
Integer === 'fourtytwo' # => false

/ell/ === 'Hello'       # => true
/ell/ === 'Foobar'      # => false
```

**Classi che `===` override `===`**

Molte classi `===` override `===` per fornire semantica significativa nelle istruzioni caso. Alcuni di loro sono:

```
╔═══════════════╦═══════════════════╗
║     Class     ║    Synonym for    ║
╠═══════════════╬═══════════════════╣
║ Array         ║ ==                ║
║               ║                   ║
║ Date          ║ ==                ║
║               ║                   ║
║ Module        ║ is_a?             ║
```

```
╟                   ╟                        ╢
╟ Object            ╟ ==                     ╢
╟                   ╟                        ╢
╟ Range             ╟ include?               ╢
╟                   ╟                        ╢
╟ Regexp            ╟ =~                     ╢
╟                   ╟                        ╢
╟ String            ╟ ==                     ╢
╚═══════════════════╩══════════════════════════╝
```

**Pratica raccomandata**

L'uso esplicito dell'operatore di uguaglianza dei casi `===` dovrebbe essere evitato. Non mette alla prova l'uguaglianza, ma piuttosto la *sussunzione* , e il suo uso può essere fonte di confusione. Il codice è più chiaro e più facile da capire quando viene utilizzato il metodo del sinonimo.

```
# Bad
Integer === 42
(1..5) === 3
/ell/ === 'Hello'

# Good, uses synonym method
42.is_a?(Integer)
(1..5).include?(3)
/ell/ =~ 'Hello'
```

## Operatore di navigazione sicura

Ruby 2.3.0 ha aggiunto l' *operatore di navigazione sicura* , `&.` . Questo operatore ha lo scopo di ridurre il paradigma `object && object.property && object.property.method` nelle istruzioni condizionali.

Ad esempio, si dispone di un oggetto `House` con una proprietà `address` e si desidera trovare lo `street_name` `address` . Per programmare questo in modo sicuro per evitare errori nil nelle vecchie versioni di Ruby, dovresti usare un codice come questo:

```
if house && house.address && house.address.street_name
  house.address.street_name
end
```

L'operatore di navigazione sicura accorcia questa condizione. Invece, puoi scrivere:

```
if house&.address&.street_name
  house.address.street_name
end
```

**Attenzione:**
L'operatore di navigazione sicura non ha *esattamente* lo stesso comportamento del condizionale concatenato. Usando il condizionale concatenato (primo esempio), il blocco `if` non verrebbe eseguito se, ad esempio, l' `address` fosse `false` . L'operatore di navigazione sicura riconosce solo i valori `nil` , ma consente valori come `false` . Se l' `address` è `false` , l'utilizzo di SNO genera un

errore:

```
house&.address&.street_name
# => undefined method `address' for false:FalseClass
```

Leggi operatori online: https://riptutorial.com/it/ruby/topic/3764/operatori

# Capitolo 53: operatori

## Examples

### Operatori di confronto

| Operatore | Descrizione |
|---|---|
| `==` | `true` se i due valori sono uguali. |
| `!=` | `true` se i due valori *non* sono uguali. |
| `<` | `true` se il valore dell'operando a sinistra è *inferiore* al valore a destra. |
| `>` | `true` se il valore dell'operando a sinistra è *maggiore* del valore a destra. |
| `>=` | `true` se il valore dell'operando a sinistra è *maggiore* **o** *uguale al* valore a destra. |
| `<=` | `true` se il valore dell'operando a sinistra è *inferiore* **o** *uguale al* valore a destra. |
| `<=>` | `0` se il valore dell'operando a sinistra è *uguale al* valore a destra,<br>`1` se il valore dell'operando a sinistra è *maggiore* del valore a destra,<br>`-1` se il valore dell'operando a sinistra è *inferiore* al valore a destra. |

### Operatori di assegnazione

## Assegnazione semplice

`=` è un compito semplice. Crea una nuova variabile locale se la variabile non è stata precedentemente referenziata.

```
x = 3
y = 4 + 5
puts "x is #{x}, y is #{y}"
```

Questo produrrà:

```
x is 3, y is 9
```

## Assegnazione parallela

Le variabili possono anche essere assegnate in parallelo, ad es. `x, y = 3, 9`. Ciò è particolarmente utile per lo scambio di valori:

```
x, y = 3, 9
x, y = y, x
puts "x is #{x}, y is #{y}"
```

Questo produrrà:

```
x is 9, y is 3
```

## Assegnazione abbreviata

È possibile combinare operatori e compiti. Per esempio:

```
x = 1
y = 2
puts "x is #{x}, y is #{y}"

x += y
puts "x is now #{x}"
```

Mostra il seguente risultato:

```
x is 1, y is 2
x is now 3
```

Varie operazioni possono essere utilizzate in compiti abbreviati:

| Operatore | Descrizione | Esempio | Equivalente a |
|---|---|---|---|
| `+=` | Aggiunge e riassegna la variabile | `x += y` | `x = x + y` |
| `-=` | Sottrae e riassegna la variabile | `x -= y` | `x = x - y` |
| `*=` | Moltiplica e riassegna la variabile | `x *= y` | `x = x * y` |
| `/=` | Divide e riassegna la variabile | `x /= y` | `x = x / y` |
| `%=` | Divide, prende il resto e riassegna la variabile | `x %= y` | `x = x % y` |
| `**=` | Calcola l'esponente e riassegna la variabile | `x **= y` | `x = x ** y` |

Leggi operatori online: https://riptutorial.com/it/ruby/topic/3766/operatori

# Capitolo 54: Operazioni su file e I / O

## Parametri

| Bandiera | Senso |
|---|---|
| "R" | Sola lettura, inizia all'inizio del file (modalità predefinita). |
| "R +" | Lettura-scrittura, inizia all'inizio del file. |
| "W" | Solo scrittura, tronca il file esistente a lunghezza zero o crea un nuovo file per la scrittura. |
| "W +" | Lettura-scrittura, tronca il file esistente a lunghezza zero o crea un nuovo file per la lettura e la scrittura. |
| "un" | Solo scrittura, inizia alla fine del file se il file esiste, altrimenti crea un nuovo file per la scrittura. |
| "A +" | Read-write, inizia alla fine del file se il file esiste, altrimenti crea un nuovo file per la lettura e la scrittura. |
| "B" | Modalità file binario. Elimina la conversione EOL <-> CRLF su Windows. E imposta la codifica esterna su ASCII-8BIT se non specificato esplicitamente. (Questo flag può apparire solo in combinazione con i flag precedenti. Ad esempio, `File.new("test.txt", "rb")` aprirà `test.txt` in modalità di `read-only` lettura come file `binary` .) |
| "T" | Modalità file di testo. (Questo flag può apparire solo in combinazione con i flag precedenti. Ad esempio, `File.new("test.txt", "wt")` aprirà `test.txt` in modalità di `write-only` come file di `text` .) |

## Examples

### Scrivere una stringa in un file

Una stringa può essere scritta in un file con un'istanza della classe `File` .

```
file = File.new('tmp.txt', 'w')
file.write("NaNaNaNa\n")
file.write('Batman!\n')
file.close
```

La classe `File` offre anche una scorciatoia per le operazioni `new` e `close` con il metodo `open` .

```
File.open('tmp.txt', 'w') do |f|
```

```
  f.write("NaNaNaNa\n")
  f.write('Batman!\n')
end
```

Per semplici operazioni di scrittura, una stringa può anche essere scritta direttamente in un file con `File.write` . **Si noti che questo sovrascriverà il file per impostazione predefinita.**

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n')
```

Per specificare una modalità diversa su `File.write` , `File.write` come valore di una chiave chiamata `mode` in un hash come un altro parametro.

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n', { mode: 'a'})
```

## Apri e chiude un file

Aprire e chiudere manualmente un file.

```
# Using new method
f = File.new("test.txt", "r") # reading
f = File.new("test.txt", "w") # writing
f = File.new("test.txt", "a") # appending

# Using open method
f = open("test.txt", "r")

# Remember to close files
f.close
```

Chiudi automaticamente un file usando un blocco.

```
f = File.open("test.txt", "r") do |f|
  # do something with file f
  puts f.read # for example, read it
end
```

## ottieni un singolo carattere di input

A differenza di `gets.chomp` questo non aspetterà una nuova riga.

La prima parte dello stdlib deve essere inclusa

```
require 'io/console'
```

Quindi è possibile scrivere un metodo di supporto:

```
def get_char
  input = STDIN.getch
  control_c_code = "\u0003"
  exit(1) if input == control_c_code
```

```
  input
end
```

E 'imporante di uscire se viene premuto il `control+c` .

## Leggendo da STDIN

```
# Get two numbers from STDIN, separated by a newline, and output the result
number1 = gets
number2 = gets
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb
## or:       $ echo -e "1\n2" | ruby a_plus_b.rb
```

## Lettura dagli argomenti con ARGV

```
number1 = ARGV[0]
number2 = ARGV[1]
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb 1 2
```

Leggi Operazioni su file e I / O online: https://riptutorial.com/it/ruby/topic/4310/operazioni-su-file-e-i---o

# Capitolo 55: OptionParser

## introduzione

OptionParser può essere utilizzato per l'analisi delle opzioni da riga di comando da `ARGV` .

## Examples

### Opzioni della riga di comando obbligatorie e facoltative

È relativamente semplice analizzare la riga di comando a mano se non si sta cercando qualcosa di troppo complesso:

```
# Naive error checking
abort('Usage: ' + $0 + ' site id ...') unless ARGV.length >= 2

# First item (site) is mandatory
site = ARGV.shift

ARGV.each do | id |
  # Do something interesting with each of the ids
end
```

Ma quando le opzioni cominciano a diventare più complicate, probabilmente dovrai usare un parser di opzioni come, beh, OptionParser :

```
require 'optparse'

# The actual options will be stored in this hash
options = {}

# Set up the options you are looking for
optparse = OptionParser.new do |opts|
  opts.banner = "Usage: #{$0} -s NAME id ..."

  opts.on("-s", "--site NAME", "Site name") do |s|
    options[:site] = s
  end

  opts.on( '-h', '--help', 'Display this screen' ) do
    puts opts
    exit
  end
end

# The parse! method also removes any options it finds from ARGV.
optparse.parse!
```

C'è anche un `parse` non distruttiva, ma è molto meno utile se `ARGV` utilizzare il resto di ciò che è in `ARGV` .

La classe OptionParser non ha un modo per applicare gli argomenti obbligatori (come `--site` in questo caso). Tuttavia, puoi eseguire il controllo dopo aver eseguito il `parse!` :

```
# Slightly more sophisticated error checking
if options[:site].nil? or ARGV.length == 0
  abort(optparse.help)
end
```

Per un gestore di opzioni obbligatorie più generico, vedere questa risposta . Nel caso in cui non sia chiaro, tutte le opzioni sono opzionali a meno che non si facciano il possibile per renderle obbligatorie.

## Valori standard

Con `OptionsParser` , è davvero facile impostare valori predefiniti. Basta precompilare l'hash in cui si memorizzano le opzioni in:

```
options = {
  :directory => ENV['HOME']
}
```

Quando definisci il parser, sovrascrive il valore predefinito se un utente fornisce un valore:

```
OptionParser.new do |opts|
  opts.on("-d", "--directory HOME", "Directory to use") do |d|
    options[:directory] = d
  end
end
```

## Descrizioni lunghe

A volte la tua descrizione può diventare piuttosto lunga. Per esempio `irb -h` elenca sull'argomento che legge:

```
  --context-mode n  Set n[0-3] to method to create Binding Object,
                    when new workspace was created
```

Non è immediatamente chiaro come supportarlo. La maggior parte delle soluzioni richiede la regolazione per rendere l'indentazione della seconda e delle seguenti linee allineata alla prima. Fortunatamente, il metodo `on` supporta più righe descrittive aggiungendole come argomenti separati:

```
  opts.on("--context-mode n",
          "Set n[0-3] to method to create Binding Object,",
          "when new workspace was created") do |n|
    optons[:context_mode] = n
  end
```

Puoi aggiungere tutte le linee di descrizione che desideri per spiegare completamente l'opzione.

Leggi OptionParser online: https://riptutorial.com/it/ruby/topic/9860/optionparser

# Capitolo 56: paragonabile

## Sintassi

- `include Comparable`
- implementare l'operatore space-ship ( `<=>` )

## Parametri

| Parametro | Dettagli |
| --- | --- |
| altro | L'istanza da paragonare al `self` |

## Osservazioni

`x <=> y` dovrebbe restituire un numero negativo se `x < y` , zero se `x == y` e un numero positivo se `x > y` .

## Examples

### Rettangolo comparabile per area

`Comparable` è uno dei moduli più popolari in Ruby. Il suo scopo è quello di fornire metodi di confronto di convenienza.

Per utilizzarlo, devi `include Comparable` e definire l'operatore spazio-nave ( `<=>` ):

```
class Rectangle
  include Comparable

  def initialize(a, b)
    @a = a
    @b = b
  end

  def area
    @a * @b
  end

  def <=>(other)
    area <=> other.area
  end
end

r1 = Rectangle.new(1, 1)
r2 = Rectangle.new(2, 2)
r3 = Rectangle.new(3, 3)
```

```
r2 >= r1 # => true
r2.between? r1, r3 # => true
r3.between? r1, r2 # => false
```

Leggi paragonabile online: https://riptutorial.com/it/ruby/topic/1485/paragonabile

# Capitolo 57: perfezionamenti

## Osservazioni

I perfezionamenti sono di tipo lessicale, il che significa che sono in vigore dal momento in cui sono attivati (con la parola chiave `using` ) fino a quando il controllo non cambia. Di solito il controllo viene modificato dalla fine di un modulo, classe o file.

## Examples

### Patch per scimmie con portata limitata

Il problema principale di Monkey Patching è che inquina l'ambito globale. Il tuo codice di lavoro è in balia di tutti i moduli che usi non calpestare le dita degli altri. La soluzione Ruby a questo è raffinatezza, che sono fondamentalmente patch scimmia in un ambito limitato.

```
module Patches
  refine Fixnum do
    def plus_one
      self + 1
    end

    def plus(num)
      self + num
    end

    def concat_one
      self.to_s + '1'
    end
  end
end

class RefinementTest
  # has access to our patches
  using Patches

  def initialize
    puts 1.plus_one
    puts 3.concat_one
  end
end

# Main scope doesn't have changes

1.plus_one
# => undefined method `plus_one' for 1:Fixnum (NoMethodError)

RefinementTest.new
# => 2
# => '31'
```

### Moduli dual-purpose (perfezionamenti o patch globali)

È buona norma applicare le patch con i perfezionamenti, ma a volte è bello caricarlo globalmente (ad esempio in fase di sviluppo o di test).

Ad esempio, per esempio, si desidera avviare una console, richiedere la libreria e quindi disporre dei metodi con patch disponibili nell'ambito globale. Non è possibile farlo con i perfezionamenti perché `using` deve essere chiamato in una definizione di classe / modulo. Ma è possibile scrivere il codice in modo tale che abbia un duplice scopo:

```
module Patch
  def patched?; true; end
  refine String do
    include Patch
  end
end

# globally
String.include Patch
"".patched? # => true

# refinement
class LoadPatch
  using Patch
  "".patched? # => true
end
```

## Affinamenti dinamici

I perfezionamenti hanno limitazioni speciali.

`refine` può essere usato solo nell'ambito di un modulo, ma può essere programmato usando `send :refine` .

`using` è più limitato. Può essere chiamato solo in una definizione di classe / modulo. Tuttavia, può accettare una variabile che punta a un modulo e può essere invocata in un ciclo.

Un esempio che mostra questi concetti:

```
module Patch
  def patched?; true; end
end

Patch.send(:refine, String) { include Patch }

patch_classes = [Patch]

class Patched
  patch_classes.each { |klass| using klass }
  "".patched? # => true
end
```

Poiché l' `using` è statico, è possibile che venga emesso un ordine di caricamento se i file di perfezionamento non vengono caricati per primi. Un modo per affrontare questo è quello di avvolgere la definizione di classi / moduli patchato in un proc. Per esempio:

```
module Patch
  refine String do
    def patched; true; end
  end
end

class Foo
end

# This is a proc since methods can't contain class definitions
create_patched_class = Proc.new do
  Foo.class_exec do
    class Bar
      using Patch
      def self.patched?; ''.patched == true; end
    end
  end
end
create_patched_class.call
Foo::Bar.patched? # => true
```

Chiamando il proc crea la classe patchato `Foo::Bar` . Questo può essere ritardato fino a dopo che tutto il codice è stato caricato.

Leggi perfezionamenti online: https://riptutorial.com/it/ruby/topic/6563/perfezionamenti

# Capitolo 58: rbenv

## Examples

### 1. Installa e gestisci le versioni di Ruby con rbenv

Il modo più semplice per installare e gestire varie versioni di Ruby con rbenv è usare il plugin ruby-build.

Prima clona il repository rbenv nella tua home directory:

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
```

Quindi clona il plugin di ruby-build:

```
$ git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
```

Assicurati che rbenv sia inizializzato nella tua sessione shell, aggiungendolo al tuo `.bash_profile` o `.zshrc` :

```
type rbenv > /dev/null
if [ "$?" = "0" ]; then
    eval "$(rbenv init -)"
fi
```

(Questo essenzialmente verifica innanzitutto se `rbenv` è disponibile e lo inizializza).

Probabilmente dovrai riavviare la sessione shell o semplicemente aprire una nuova finestra di Terminale.

**Nota:** se stai utilizzando OSX, dovrai anche installare gli strumenti della riga di comando di Mac OS con:

```
$ xcode-select --install
```

Puoi anche installare `rbenv` usando Homebrew invece di creare dalla sorgente:

```
$ brew update
$ brew install rbenv
```

Quindi seguire le istruzioni fornite da:

```
$ rbenv init
```

**Installa una nuova versione di Ruby:**

Elenca le versioni disponibili con:

```
$ rbenv install --list
```

Scegli una versione e installala con:

```
$ rbenv install 2.2.0
```

Contrassegna la versione installata come versione globale, ovvero quella che il tuo sistema utilizza per impostazione predefinita:

```
$ rbenv global 2.2.0
```

Verifica qual è la tua versione globale:

```
$ rbenv global
=> 2.2.0
```

È possibile specificare una versione del progetto locale con:

```
$ rbenv local 2.1.2
=> (Creates a .ruby-version file at the current directory with the specified version)
```

---

Note:

[1]: Comprensione del PERCORSO

### Disinstallare un Ruby

Esistono due modi per disinstallare una versione specifica di Ruby. Il modo più semplice è semplicemente rimuovere la directory da `~/.rbenv/versions` :

```
$ rm -rf ~/.rbenv/versions/2.1.0
```

In alternativa, puoi usare il comando uninstall, che fa esattamente la stessa cosa:

```
$ rbenv uninstall 2.1.0
```

Se questa versione sembra essere in uso da qualche parte, è necessario aggiornare la versione globale o locale. Per ripristinare la versione che è la prima nel tuo percorso (di solito il valore predefinito fornito dal tuo sistema) usa:

```
$ rbenv global system
```

Leggi rbenv online: https://riptutorial.com/it/ruby/topic/4096/rbenv

# Capitolo 59: Ricevitori impliciti e Sé comprensivo

## Examples

### C'è sempre un ricevitore implicito

In Ruby, c'è sempre un ricevitore implicito per tutte le chiamate di metodo. La lingua mantiene un riferimento all'attuale ricevitore implicito memorizzato nella variabile `self` . Certe parole chiave del linguaggio come la `class` e `module` cambierà quello `self` punti a. Comprendere questi comportamenti è molto utile per padroneggiare la lingua.

Ad esempio, quando apri per la prima volta `irb`

```
irb(main):001:0> self
=> main
```

In questo caso l'oggetto `main` è il ricevitore implicito (vedi http://stackoverflow.com/a/917842/417872 per ulteriori informazioni su `main` ).

È possibile definire i metodi sul ricevitore implicito utilizzando la parola chiave `def` . Per esempio:

```
irb(main):001:0> def foo(arg)
irb(main):002:1> arg.to_s
irb(main):003:1> end
=> :foo
irb(main):004:0> foo 1
=> "1"
```

Questo ha definito il metodo foo sull'istanza dell'oggetto principale in esecuzione nel proprio repl.

Si noti che le variabili locali vengono controllate prima dei nomi dei metodi, in modo tale che se si definisce una variabile locale con lo stesso nome, il suo riferimento sostituirà il riferimento al metodo. Continuando dall'esempio precedente:

```
irb(main):005:0> defined? foo
=> "method"
irb(main):006:0> foo = 1
=> 1
irb(main):007:0> defined? foo
=> "local-variable"
irb(main):008:0> foo
=> 1
irb(main):009:0> method :foo
=> #<Method: Object#foo>
```

Il `method` method può ancora trovare il metodo `foo` perché non controlla le variabili locali, mentre il normale `foo` riferimento lo fa.

## Le parole chiave cambiano il ricevitore implicito

Quando si definisce una classe o un modulo, il ricevitore implicito diventa un riferimento alla classe stessa. Per esempio:

```
puts "I am #{self}"
class Example
  puts "I am #{self}"
end
```

Eseguendo il codice sopra verrà stampato:

```
"I am main"
"I am Example"
```

## Quando usare se stessi?

La maggior parte del codice Ruby utilizza il ricevitore implicito, quindi i programmatori che sono nuovi a Ruby sono spesso confusi su quando utilizzare `self` . La risposta pratica è che il `self` è usato in due modi principali:

**1. Per cambiare il ricevitore.**

Ordinariamente il comportamento di `def` all'interno di una classe o di un modulo è quello di creare metodi di istanza. Il sé può essere usato per definire i metodi sulla classe.

```
class Foo
  def bar
    1
  end

  def self.bar
    2
  end
end

Foo.new.bar #=> 1
Foo.bar #=> 2
```

**2. Disambiguare il ricevitore**

Quando le variabili locali possono avere lo stesso nome di un metodo, può essere richiesto un ricevitore esplicito per disambiguare.

Esempi:

```
class Example
  def foo
    1
  end

  def bar
```

```
    foo + 1
  end

  def baz(foo)
    self.foo + foo # self.foo is the method, foo is the local variable
  end

  def qux
    bar = 2
    self.bar + bar # self.bar is the method, bar is the local variable
  end
end

Example.new.foo    #=> 1
Example.new.bar    #=> 2
Example.new.baz(2) #=> 3
Example.new.qux    #=> 4
```

L'altro caso comune che richiede disambiguazione implica metodi che finiscono nel segno di uguaglianza. Per esempio:

```
class Example
  def foo=(input)
    @foo = input
  end

  def get_foo
    @foo
  end

  def bar(input)
    foo = input # will create a local variable
  end

  def baz(input)
    self.foo = input # will call the method
  end
end

e = Example.new
e.get_foo #=> nil
e.foo = 1
e.get_foo #=> 1
e.bar(2)
e.get_foo #=> 1
e.baz(2)
e.get_foo #=> 2
```

Leggi Ricevitori impliciti e Sé comprensivo online:
https://riptutorial.com/it/ruby/topic/5856/ricevitori-impliciti-e-se-comprensivo

# Capitolo 60: Ricorsione in Ruby

## Examples

### Funzione ricorsiva

Iniziamo con un semplice algoritmo per vedere come la ricorsione potrebbe essere implementata in Ruby.

Un panificio ha prodotti da vendere. I prodotti sono in confezioni. Serve solo ordini in pacchetti. L'imballaggio inizia dalla confezione più grande e le quantità rimanenti vengono riempite dalle confezioni successive disponibili.

Ad esempio, se viene ricevuto un ordine di 16, la panetteria assegna 2 pacchetti da 5 e 2 pacchetti da 3. 2 *5* + *2* 3 = 16. Vediamo come questo è implementato in ricorsione. "allocare" è la funzione ricorsiva qui.

```
#!/usr/bin/ruby

class Bakery
  attr_accessor :selected_packs

  def initialize
    @packs = [5,3] # pack sizes 5 and 3
    @selected_packs = []
  end

  def allocate(qty)
    remaining_qty = nil

    # ============================================
    # packs are allocated in large packs first order
    # to minimize the packaging space
    # ============================================
    @packs.each do |pack|
      remaining_qty = qty - pack

      if remaining_qty > 0
        ret_val = allocate(remaining_qty)
        if ret_val == 0
          @selected_packs << pack
          remaining_qty = 0
          break
        end
      elsif remaining_qty == 0
        @selected_packs << pack
        break
      end
    end

    remaining_qty
  end
end
```

```
bakery = Bakery.new
bakery.allocate(16)
puts "Pack combination is: #{bakery.selected_packs.inspect}"
```

L'output è:

> La combinazione di pacchetti è: [3, 3, 5, 5]

## Ricorsione di coda

Molti algoritmi ricorsivi possono essere espressi usando l'iterazione. Ad esempio, la massima funzione denominatore comune può essere scritta in modo ricorsivo :

```
def gdc (x, y)
  return x if y == 0
  return gdc(y, x%y)
end
```

o iterativamente:

```
def gdc_iter (x, y)
  while y != 0 do
    x, y = y, x%y
  end

  return x
end
```

I due algoritmi sono equivalenti in teoria, ma la versione ricorsiva rischia un SystemStackError . Tuttavia, poiché il metodo ricorsivo termina con una chiamata a se stesso, potrebbe essere ottimizzato per evitare un overflow dello stack. Un altro modo per dirla: l'algoritmo ricorsivo può generare lo stesso codice macchina del iterativo *se* il compilatore sa cercare la chiamata al metodo ricorsiva alla fine del metodo. Ruby non ottimizza le chiamate tail per impostazione predefinita, ma puoi attivarlo con :

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
```

Oltre a attivare l'ottimizzazione della coda, è necessario disattivare anche la traccia delle istruzioni. Sfortunatamente, queste opzioni si applicano solo al momento della compilazione, quindi è necessario `require` il metodo ricorsivo da un altro file o `eval` la definizione del metodo:

```
RubyVM::InstructionSequence.new(<<-EOF).eval
  def me_myself_and_i
    me_myself_and_i
  end
EOF
me_myself_and_i # Infinite loop, not stack overflow
```

Infine, la chiamata di ritorno finale deve restituire il metodo e *solo il metodo* . Ciò significa che dovrai riscrivere la funzione fattoriale standard:

```
def fact(x)
  return 1 if x <= 1
  return x*fact(x-1)
end
```

Per qualcosa come:

```
 def fact(x, acc=1)
   return acc if x <= 1
   return fact(x-1, x*acc)
 end
```

Questa versione passa la somma accumulata tramite un secondo argomento (facoltativo) che assume come valore predefinito 1.

Ulteriori letture: Ottimizzazione chiamata coda in Ruby e Tailin 'Ruby .

Leggi Ricorsione in Ruby online: https://riptutorial.com/it/ruby/topic/7986/ricorsione-in-ruby

# Capitolo 61: Ruby Version Manager

## Examples

### Come creare gemset

Per creare un gemset abbiamo bisogno di creare un file `.rvmrc` .

**Sintassi:**

```
$ rvm --rvmrc --create <ruby-version>@<gemsetname>
```

**Esempio:**

```
$ rvm --rvmrc --create ruby-2.2.2@myblog
```

La riga sopra creerà un file `.rvmrc` nella directory principale dell'app.

Per ottenere l'elenco dei gemset disponibili, utilizzare il seguente comando:

```
$ rvm list gemsets
```

### Installare Ruby con RVM

*Ruby Version Manager* è uno strumento da riga di comando per installare e gestire semplicemente diverse versioni di Ruby.

- `rvm istall 2.3.1` per esempio installa Ruby versione 2.3.1 sul tuo computer.
- Con la `rvm list` puoi vedere quali versioni sono installate e quali sono effettivamente impostate per l'uso.

  ```
  user@dev:~$ rvm list

  rvm rubies

  =* ruby-2.3.1 [ x86_64 ]

  # => - current
  # =* - current && default
  #  * - default
  ```

- Con `rvm use 2.3.0` puoi cambiare tra le versioni installate.

Leggi Ruby Version Manager online: https://riptutorial.com/it/ruby/topic/4040/ruby-version-manager

# Capitolo 62: simboli

## Sintassi

- :simbolo
- :'simbolo'
- :"simbolo"
- .to_sym "simbolo"
- % s {simbolo}

## Osservazioni

## Vantaggi dell'uso di simboli su stringhe:

### 1. Un simbolo Ruby è un oggetto con confronto O (1)

Per confrontare due stringhe, è potenzialmente necessario esaminare ogni carattere. Per due stringhe di lunghezza N, ciò richiederà N + 1 confronti

```
def string_compare str1, str2
  if str1.length != str2.length
        return false
  end
  for i in 0...str1.length
    return false if str1[i] != str2[i]
  end
  return true
end
string_compare "foobar", "foobar"
```

Ma poiché ogni aspetto di: foobar si riferisce allo stesso oggetto, possiamo confrontare i simboli osservando gli ID oggetto. Possiamo farlo con un solo confronto. (O (1))

```
def symbol_compare sym1, sym2
  sym1.object_id == sym2.object_id
end
symbol_compare :foobar, :foobar
```

### 2. Un simbolo Ruby è un'etichetta in un'enumerazione in formato libero

In C ++, possiamo usare "enumerazioni" per rappresentare famiglie di costanti correlate:

```
enum BugStatus { OPEN, CLOSED };
BugStatus original_status = OPEN;
BugStatus current_status  = CLOSED;
```

Ma poiché Ruby è un linguaggio dinamico, non ci preoccupiamo di dichiarare un tipo di BugStatus o di tenere traccia dei valori legali. Invece, rappresentiamo i valori di enumerazione come simboli:

```
original_status = :open
current_status  = :closed
```

**3. Un simbolo di Ruby è un nome costante e univoco**

In Ruby, possiamo cambiare il contenuto di una stringa:

```
"foobar"[0] = ?b # "boo"
```

Ma non possiamo cambiare il contenuto di un simbolo:

```
:foobar[0]  = ?b # Raises an error
```

**4. Un simbolo di Ruby è la parola chiave per un argomento di parole chiave**

Quando si passano gli argomenti delle parole chiave a una funzione Ruby, si specificano le parole chiave usando i simboli:

```
# Build a URL for 'bug' using Rails.
url_for :controller => 'bug',
        :action => 'show',
        :id => bug.id
```

**5. Un simbolo rubino è una scelta eccellente per un tasto cancelletto**

In genere, utilizzeremo i simboli per rappresentare le chiavi di una tabella hash:

```
options = {}
options[:auto_save]     = true
options[:show_comments] = false
```

# Examples

## Creare un simbolo

Il modo più comune per creare un oggetto `Symbol` è il prefisso dell'identificatore di stringa con due punti:

```
:a_symbol       # => :a_symbol
:a_symbol.class # => Symbol
```

Ecco alcuni modi alternativi per definire un `Symbol` , in combinazione con un letterale `String` :

```
:"a_symbol"
"a_symbol".to_sym
```

I simboli hanno anche una sequenza `%s` che supporta delimitatori arbitrari simili a come `%q` e `%Q` funzionano per le stringhe:

```
%s(a_symbol)
%s{a_symbol}
```

Il `%s` è particolarmente utile per creare un simbolo da un input che contiene uno spazio bianco:

```
%s{a symbol} # => :"a symbol"
```

Mentre alcuni simboli interessanti ( `:/` `:[]` `:^` , ecc.) Possono essere creati con determinati identificatori di stringa, si noti che i simboli non possono essere creati usando un identificatore numerico:

```
:1 # => syntax error, unexpected tINTEGER, ...
:0.3 # => syntax error, unexpected tFLOAT, ...
```

I simboli possono finire con un singolo `?` o `!` senza bisogno di usare una stringa letterale come identificativo del simbolo:

```
:hello?  # :"hello?" is not necessary.
:world!  # :"world!" is not necessary.
```

Nota che tutti questi diversi metodi di creazione dei simboli restituiscono lo stesso oggetto:

```
:symbol.object_id == "symbol".to_sym.object_id
:symbol.object_id == %s{symbol}.object_id
```

Dal momento che Ruby 2.0 c'è una scorciatoia per creare una serie di simboli dalle parole:

```
%i(numerator denominator) == [:numerator, :denominator]
```

## Conversione di una stringa in un simbolo

Dato una `String` :

```
s = "something"
```

ci sono diversi modi per convertirlo in un `Symbol` :

```
s.to_sym
# => :something
:"#{s}"
# => :something
```

## Conversione di un simbolo in stringa

Dato un `Symbol` :

```
s = :something
```

Il modo più semplice per convertirlo in una `String` consiste nell'utilizzare il metodo `Symbol#to_s` :

```
s.to_s
# => "something"
```

Un altro modo per farlo è utilizzare il metodo `Symbol#id2name` , che è un alias per il metodo `Symbol#to_s` . Ma è un metodo unico per la classe `Symbol` :

```
s.id2name
# => "something"
```

Leggi simboli online: https://riptutorial.com/it/ruby/topic/873/simboli

# Capitolo 63: Singleton Class

## Sintassi

- singleton_class = class << object; fine di sé

## Osservazioni

Le classi Singleton hanno solo un'istanza: il loro oggetto corrispondente. Questo può essere verificato interrogando Ruby's `ObjectSpace` :

```
instances = ObjectSpace.each_object object.singleton_class

instances.count            # => 1
instances.include? object  # => true
```

Usando `<` , possono anche essere verificate come sottoclassi della classe effettiva dell'oggetto:

```
object.singleton_class < object.class  # => true
```

---

Riferimenti:

- Tre contesti impliciti in Ruby

## Examples

### introduzione

Ruby ha tre tipi di oggetti:

- Classi e moduli che sono istanze di classe Classe o modulo di classe.
- Istanze di classi
- Singleton Classes.

Ogni oggetto ha una classe che contiene i suoi metodi:

```
class Example
end

object = Example.new

object.class  # => Example
Example.class # => Class
Class.class   # => Class
```

Gli oggetti stessi non possono contenere metodi, solo la loro classe può. Ma con le classi

singleton, è possibile aggiungere metodi a qualsiasi oggetto incluse altre classi di singleton.

```
def object.foo
  :foo
end
object.foo #=> :foo
```

`foo` è definito sulla classe di `object` singleton. Altri `Example` casi non possono rispondere a `foo` .

Ruby crea corsi singoli su richiesta. Accedervi o aggiungere metodi a loro forza costringe Ruby a crearli.

## Accesso alla classe Singleton

Esistono due modi per ottenere la classe singleton di un oggetto

- metodo `singleton_class` .
- Riapertura della classe singleton di un oggetto e ritorno di `self` .

```
object.singleton_class
```

---

```
singleton_class = class << object
  self
end
```

## Accesso alle variabili istanza / classe nelle classi Singleton

Le classi Singleton condividono le loro istanze / variabili di classe con il loro oggetto.

```
class Example
  @@foo = :example
end

def Example.foo
  class_variable_get :@@foo
end

Example.foo #=> :example
```

---

```
class Example
  def initialize
    @foo = 1
  end

  def foo
    @foo
  end
end

e = Example.new
```

```
e.instance_eval <<-BLOCK
  def self.increase_foo
    @foo += 1
  end
BLOCK

e.increase_foo
e.foo #=> 2
```

---

I blocchi si chiudono attorno alla loro variabile di istanza / classe target. Non è possibile accedere alle variabili di istanza o classe utilizzando un blocco in `class_eval` o `instance_eval` . Passare una stringa a `class_eval` o utilizzare `class_variable_get` il problema.

```
class Foo
  @@foo = :foo
end

class Example
  @@foo = :example

  Foo.define_singleton_method :foo do
    @@foo
  end
end

Foo.foo #=> :example
```

### Eredità della classe Singleton

## Sottoclassi anche sottoclassi Singleton Class

```
class Example
end

Example.singleton_class #=> #<Class:Example>

def Example.foo
  :example
end

class SubExample < Example
end

SubExample.foo #=> :example

SubExample.singleton_class.superclass #=> #<Class:Example>
```

## L'estensione o l'inclusione di un modulo non estende la classe Singleton

```
module ExampleModule
end
```

```
def ExampleModule.foo
  :foo
end

class Example
  extend ExampleModule
  include ExampleModule
end

Example.foo #=> NoMethodError: undefined method
```

## Propagazione dei messaggi con la classe Singleton

Le istanze non contengono mai un metodo che portano solo dati. Tuttavia possiamo definire una classe singleton per qualsiasi oggetto che includa un'istanza di una classe.

Quando un messaggio viene passato a un oggetto (il metodo viene chiamato) Ruby controlla prima se una classe singleton è definita per quell'oggetto e se può rispondere a quel messaggio altrimenti Ruby verifica la catena di antenati della classe dell'istanza e ne va su.

```
class Example
  def foo
    :example
  end
end

Example.new.foo #=> :example

module PrependedModule
  def foo
    :prepend
  end
end

class Example
  prepend PrependedModule
end

Example.ancestors #=> [Prepended, Example, Object, Kernel, BasicObject]
e = Example.new
e.foo #=> :prepended

def e.foo
  :singleton
end

e.foo #=> :singleton
```

## Riapertura (scimmia rattoppando) Singleton Classes

Esistono tre modi per riaprire una classe Singleton

- Usando `class_eval` su una classe singleton.
- Usando il `class <<` block.

- Utilizzando `def` per definire direttamente un metodo sulla classe Singleton dell'oggetto

```
class Example
end

Example.singleton_class.class_eval do
  def foo
    :foo
  end
end

Example.foo #=> :foo
```

---

```
class Example
end

class << Example
  def bar
    :bar
  end
end

Example.bar #=> :bar
```

---

```
class Example
end

def Example.baz
  :baz
end

Example.baz #=> :baz
```

Ogni oggetto ha una classe singleton a cui è possibile accedere

```
class Example
end
ex1 = Example.new
def ex1.foobar
  :foobar
end
ex1.foobar #=> :foobar

ex2 = Example.new
ex2.foobar #=> NoMethodError
```

## Lezioni di Singleton

Tutti gli oggetti sono istanze di una classe. Tuttavia, non è tutta la verità. In Ruby, ogni oggetto ha anche una *classe singleton* un po 'nascosta.

Questo è ciò che consente di definire i metodi su singoli oggetti. La classe singleton si trova tra l'oggetto stesso e la sua classe effettiva, quindi tutti i metodi definiti su di esso sono disponibili per

quell'oggetto e solo quell'oggetto.

```
object = Object.new

def object.exclusive_method
  'Only this object will respond to this method'
end

object.exclusive_method
# => "Only this object will respond to this method"

Object.new.exclusive_method rescue $!
# => #<NoMethodError: undefined method `exclusive_method' for #<Object:0xa17b77c>>
```

L'esempio precedente potrebbe essere stato scritto usando `define_singleton_method` :

```
object.define_singleton_method :exclusive_method do
  "The method is actually defined in the object's singleton class"
end
```

Il che equivale a definire il metodo sulla `singleton_class` `object` :

```
# send is used because define_method is private
object.singleton_class.send :define_method, :exclusive_method do
  "Now we're defining an instance method directly on the singleton class"
end
```

Prima dell'esistenza di `singleton_class` come parte dell'API core di Ruby, le classi singleton erano conosciute come *metaclassi* e potevano essere accessibili tramite il seguente idioma:

```
class << object
  self  # refers to object's singleton_class
end
```

Leggi Singleton Class online: https://riptutorial.com/it/ruby/topic/4277/singleton-class

# Capitolo 64: Sistema operativo o comandi Shell

## introduzione

Esistono molti modi per interagire con il sistema operativo. Dall'interno di Ruby è possibile eseguire comandi shell o di sistema o processi secondari.

## Osservazioni

**Exec:**
Exec ha funzionalità molto limitate e quando viene eseguito uscirà dal programma Ruby ed eseguirà il comando.

**Il comando di sistema:**
Il comando di sistema viene eseguito in una sotto-shell invece di sostituire il processo corrente e restituisce true o nill. Il comando di sistema è, come i backtick, un'operazione di blocco in cui l'applicazione principale attende fino al completamento del risultato dell'operazione di sistema. Qui l'operazione principale non deve mai preoccuparsi di catturare un'eccezione sollevata dal processo figlio.

L'output della funzione di sistema sarà sempre vero o nullo a seconda che lo script sia stato eseguito senza errori. Pertanto, ogni errore durante l'esecuzione dello script non verrà passato alla nostra applicazione. L'operazione principale non deve mai preoccuparsi di acquisire un'eccezione generata dal processo figlio. In questo caso l'output è nullo perché il processo figlio ha generato un'eccezione.
Questa è un'operazione di blocco in cui il programma Ruby attenderà fino al completamento dell'operazione del comando prima di procedere.
L'operazione di sistema utilizza fork per eseguire il fork del processo corrente e quindi esegue l'operazione specificata utilizzando exec.

**I backtick (`):**
Il carattere backtick è solitamente posizionato sotto il tasto escape sulla tastiera. I backtick vengono eseguiti in una sotto-shell invece di sostituire il processo corrente e restituiscono il risultato del comando.
Qui possiamo ottenere l'output del comando ma il programma si bloccherà quando viene generata un'eccezione.
Se esiste un'eccezione nel processo secondario, tale eccezione viene fornita al processo principale e il processo principale potrebbe terminare se l'eccezione non viene gestita. Questa è un'operazione di blocco in cui il programma Ruby attenderà fino al completamento dell'operazione del comando prima di procedere.
L'operazione di sistema utilizza fork per eseguire il fork del processo corrente e quindi esegue l'operazione specificata utilizzando exec.

**IO.popen:**
IO.popen viene eseguito in un processo secondario. Qui l'input standard del processo secondario e l'uscita standard sono collegati all'oggetto IO.

**Popen3:**
Popen3 ti consente di accedere allo standard input, allo standard output e all'errore standard.
Gli input e output standard del sottoprocesso verranno restituiti in oggetti IO.

**$? (come $ CHILD_STATUS)**
Può essere utilizzato con le operazioni backtick, system () o% x {} e fornirà lo stato dell'ultimo comando eseguito dal sistema.
Questo potrebbe essere utile per accedere a `exitstatus` e alle proprietà `pid` .

```
$?.exitstatus
```

# Examples

## Metodi consigliati per eseguire il codice shell in Ruby:

**Open3.popen3 o Open3.capture3:**
Open3 in realtà utilizza solo il comando spawn di Ruby, ma offre un'API molto migliore.

**Open3.popen3**

Popen3 viene eseguito in un sottoprocesso e restituisce stdin, stdout, stderr e wait_thr.

```
require 'open3'
stdin, stdout, stderr, wait_thr = Open3.popen3("sleep 5s && ls")
puts "#{stdout.read} #{stderr.read} #{wait_thr.value.exitstatus}"
```

o

```
require 'open3'
cmd = 'git push heroku master'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  puts "stdout is:" + stdout.read
  puts "stderr is:" + stderr.read
end
```

uscirà: **stdout è: stderr è: fatale: non un repository git (o nessuna delle directory madri): .git**

o

```
require 'open3'
cmd = 'ping www.google.com'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  while line = stdout.gets
    puts line
  end
end
```

produrrà:

**Pinging www.google.com [216.58.223.36] con 32 byte di dati:**
**Risposta da 216.58.223.36: byte = 32 tempo = 16 ms TTL = 54**
**Risposta da 216.58.223.36: byte = 32 tempo = 10 ms TTL = 54**
**Risposta da 216.58.223.36: byte = 32 tempo = 21 ms TTL = 54**
**Risposta da 216.58.223.36: byte = 32 tempo = 29 ms TTL = 54**
**Statistiche ping per 216.58.223.36:**
**Pacchetti: Inviato = 4, Ricevuto = 4, Perso = 0 (0% di perdita),**
**Tempi approssimativi di andata e ritorno in millisecondi:**
**Minimo = 10 ms, Massimo = 29 ms, Medio = 19 ms**

---

**Open3.capture3:**

```
require 'open3'

stdout, stderr, status = Open3.capture3('my_funky_command', 'and', 'some', 'argumants')
if status.success?
  # command completed successfully, do some more stuff
else
  raise "An error occured"
end
```

o

```
Open3.capture3('/some/binary with some args')
```

Non consigliato però, a causa di sovraccarico aggiuntivo e il potenziale per iniezioni di shell.

Se il comando legge da stdin e vuoi nutrirlo con alcuni dati:

```
Open3.capture3('my_funky_command', stdin_data: 'read from stdin')
```

Esegui il comando con una diversa directory di lavoro, usando chdir:

```
Open3.capture3('my_funky_command', chdir: '/some/directory')
```

## Modi classici per eseguire il codice shell in Ruby:

**Exec:**

```
exec 'echo "hello world"'
```

o

```
exec ('echo "hello world"')
```

**Il comando di sistema:**

```
system 'echo "hello world"'
```

Uscirà "ciao mondo" nella finestra di comando.

o

```
system ('echo "hello world"')
```

Il comando di sistema può restituire un vero se il comando ha avuto successo o nill quando no.

```
result = system 'echo "hello world"'
puts result  # will return a true in the command window
```

**I backtick (`):**

`echo "hello world"` Apparirà "Hello World" nella finestra di comando.

Puoi anche prendere il risultato.

```
result = `echo "hello world"`
puts "We always code a " + result
```

**IO.popen:**

```
# Will get and return the current date from the system
IO.popen("date") { |f| puts f.gets }
```

Leggi Sistema operativo o comandi Shell online: https://riptutorial.com/it/ruby/topic/10921/sistema-operativo-o-comandi-shell

# Capitolo 65: stringhe

## Sintassi

- 'Una stringa' // crea una stringa tramite letterale con quotatura singola
- "Una stringa" // crea una stringa tramite letterale a virgolette doppie
- String.new ("Una stringa")
- % q (A string) // sintassi alternativa per la creazione di stringhe con quotatura singola
- % Q (A string) // sintassi alternativa per la creazione di stringhe con doppie virgolette

## Examples

### Differenza tra valori letterali stringa a virgolette singole e virgolette

La differenza principale è che i letterali `String` virgolette doppie supportano le interpolazioni di stringhe e il set completo di sequenze di escape.

Ad esempio, possono includere espressioni Ruby arbitrarie tramite interpolazione:

```
# Single-quoted strings don't support interpolation
puts 'Now is #{Time.now}'
# Now is #{Time.now}

# Double-quoted strings support interpolation
puts "Now is #{Time.now}"
# Now is 2016-07-21 12:43:04 +0200
```

Le stringhe con doppia citazione supportano anche l' intera serie di sequenze di escape, tra cui `"\n"` , `"\t"` ...

```
puts 'Hello\nWorld'
# Hello\nWorld

puts "Hello\nWorld"
# Hello
# World
```

... mentre le stringhe a virgolette singole *non* supportano sequenze di escape, è utile il set minimo necessario per le stringhe con quotatura singola: virgolette singole letterali e barre rovesciate, rispettivamente `'\''` e `'\\'` .

### Creare una stringa

Ruby offre diversi modi per creare un oggetto `String` . Il modo più comune consiste nell'utilizzare virgolette singole o doppie per creare una " stringa letterale ":

```
s1 = 'Hello'
```

```
s2 = "Hello"
```

La differenza principale è che i letterali stringa con virgolette doppie sono un po 'più flessibili in quanto supportano l'interpolazione e alcune sequenze di escape backslash.

Ci sono anche molti altri modi possibili per creare una stringa letterale usando delimitatori di stringhe arbitrari. Un delimitatore di stringhe arbitrario è un `%` seguito da una coppia di delimitatori corrispondente:

```
%(A string)
%{A string}
%<A string>
%|A string|
%!A string!
```

Infine, puoi usare la sequenza `%q` e `%Q` , che sono equivalenti a `'` e `"` ":

```
puts %q(A string)
# A string
puts %q(Now is #{Time.now})
# Now is #{Time.now}

puts %Q(A string)
# A string
puts %Q(Now is #{Time.now})
# Now is 2016-07-21 12:47:45 +0200
```

`%q` sequenze `%q` e `%Q` sono utili quando la stringa contiene virgolette singole, virgolette doppie o un mix di entrambi. In questo modo, non è necessario sfuggire al contenuto:

```
%Q(<a href="/profile">User's profile<a>)
```

Puoi utilizzare diversi delimitatori, purché vi sia una coppia corrispondente:

```
%q(A string)
%q{A string}
%q<A string>
%q|A string|
%q!A string!
```

## Concatenazione di stringhe

Stringhe concatenate con l'operatore `+` :

```
s1 = "Hello"
s2 = " "
s3 = "World"

puts s1 + s2 + s3
# => Hello World

s = s1 + s2 + s3
```

```
puts s
# => Hello World
```

O con l'operatore `<<` :

```
s = 'Hello'
s << ' '
s << 'World'
puts s
# => Hello World
```

Si noti che l'operatore `<<` modifica l'oggetto sul lato sinistro.

Puoi anche moltiplicare le stringhe, ad es

```
"wow" * 3
# => "wowwowwow"
```

## Interpolazione a stringa

Il delimitatore a virgolette doppie `"` e `%Q` sequenza `%Q` supporta l'interpolazione delle stringhe usando `#{ruby_expression}` :

```
puts "Now is #{Time.now}"
# Now is Now is 2016-07-21 12:47:45 +0200

puts %Q(Now is #{Time.now})
# Now is Now is 2016-07-21 12:47:45 +0200
```

## Manipolazione del caso

```
"string".upcase     # => "STRING"
"STRING".downcase   # => "string"
"String".swapcase   # => "sTRING"
"string".capitalize # => "String"
```

Questi quattro metodi non modificano il ricevitore originale. Per esempio,

```
str = "Hello"
str.upcase # => "HELLO"
puts str   # => "Hello"
```

Esistono quattro metodi simili che eseguono le stesse azioni ma modificano il ricevitore originale.

```
"string".upcase!     # => "STRING"
"STRING".downcase!   # => "string"
"String".swapcase!   # => "sTRING"
"string".capitalize! # => "String"
```

Per esempio,

```
str = "Hello"
str.upcase!  # => "HELLO"
puts str     # => "HELLO"
```

Gli appunti:

- prima di Ruby 2.4 questi metodi non gestiscono unicode.

## Divisione di una stringa

`String#split` divide una `String` in una `Array` , in base a un delimitatore.

```
"alpha,beta".split(",")
# => ["alpha", "beta"]
```

Una `String` vuota risulta in una `Array` vuota:

```
"".split(",")
# => []
```

Un delimitatore non corrispondente risulta in una `Array` contenente un singolo elemento:

```
"alpha,beta".split(".")
# => ["alpha,beta"]
```

Puoi anche dividere una stringa usando le espressioni regolari:

```
"alpha, beta,gamma".split(/, ?/)
# => ["alpha", "beta", "gamma"]
```

Il delimitatore è facoltativo, per impostazione predefinita una stringa viene divisa in spazi vuoti:

```
"alpha beta".split
# => ["alpha", "beta"]
```

## Unione di stringhe

`Array#join` unisce a una `Array` in una `String` , in base a un delimitatore:

```
["alpha", "beta"].join(",")
# => "alpha,beta"
```

Il delimitatore è facoltativo e il valore predefinito è una `String` vuota.

```
["alpha", "beta"].join
# => "alphabeta"
```

Una `Array` vuota risulta in una `String` vuota, indipendentemente dal delimitatore utilizzato.

```
[].join(",")
# => ""
```

## Stringhe multilinea

Il modo più semplice per creare una stringa multilinea è utilizzare semplicemente più righe tra virgolette:

```
address = "Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the
proposition that all men are created equal."
```

Il problema principale di questa tecnica è che se la stringa include una citazione, interromperà la sintassi della stringa. Per ovviare al problema, puoi utilizzare invece un heredoc :

```
puts <<-RAVEN
  Once upon a midnight dreary, while I pondered, weak and weary,
  Over many a quaint and curious volume of forgotten lore—
      While I nodded, nearly napping, suddenly there came a tapping,
  As of some one gently rapping, rapping at my chamber door.
  "'Tis some visitor," I muttered, "tapping at my chamber door—
              Only this and nothing more."
  RAVEN
```

Ruby supporta i documenti in stile shell qui con `<<EOT` , ma il testo che chiude deve iniziare la riga. Ciò rovina il rientro del codice, quindi non c'è un motivo in più per usare questo stile. Sfortunatamente, la stringa avrà indentazioni a seconda di come il codice stesso sia rientrato.

Ruby 2.3 risolve il problema introducendo `<<~` che rimuove gli spazi in eccesso:

2.3

```
def build_email(address)
  return (<<~EMAIL)
    TO: #{address}

    To Whom It May Concern:

    Please stop playing the bagpipes at sunrise!

    Regards,
    Your neighbor
  EMAIL
end
```

Le stringhe percentuali funzionano anche per creare stringhe multilinea:

```
%q(
HAMLET        Do you see yonder cloud that's almost in shape of a camel?
POLONIUS        By the mass, and 'tis like a camel, indeed.
HAMLET        Methinks it is like a weasel.
POLONIUS        It is backed like a weasel.
HAMLET        Or like a whale?
```

```
POLONIUS        Very like a whale
)
```

Esistono alcuni modi per evitare l'interpolazione e le sequenze di escape:

- Virgolette singole invece di virgolette doppie: `'\n is a carriage return.'`
- Minuscolo `q` in una stringa percentuale: `%q[#{not-a-variable}]`
- Cita solo la stringa terminale in un heredoc:

```
<<-'CODE'
   puts 'Hello world!'
CODE
```

## Stringhe formattate

Ruby può iniettare una matrice di valori in una stringa sostituendo qualsiasi segnaposto con i valori dell'array fornito.

```
"Hello %s, my name is %s!" % ['World', 'br3nt']
# => Hello World, my name is br3nt!
```

I segnaposto sono rappresentati da due `%s` e i valori sono forniti dall'array `['Hello', 'br3nt']` . L'operatore `%` ordina alla stringa di iniettare i valori dell'array.

## Sostituzioni di carattere stringa

Il metodo `tr` restituisce una copia di una stringa in cui i caratteri del primo argomento sono sostituiti dai caratteri del secondo argomento.

```
"string".tr('r', 'l') # => "stling"
```

Per sostituire solo la prima occorrenza di un motivo con un'altra espressione, utilizzare il metodo `sub`

```
"string ring".sub('r', 'l') # => "stling ring"
```

Se si desidera sostituire *tutte le* occorrenze di un modello con tale espressione, utilizzare `gsub`

```
"string ring".gsub('r','l') # => "stling ling"
```

Per cancellare i caratteri, passa una stringa vuota per il secondo parametro

Puoi anche usare espressioni regolari in tutti questi metodi.

È importante notare che questi metodi restituiranno solo una nuova copia di una stringa e non modificheranno la stringa sul posto. Per farlo, devi usare il `tr!` , `sub!` e `gsub!` metodi rispettivamente.

## Capire i dati in una stringa

In Ruby, una stringa è solo una sequenza di byte insieme al nome di una codifica (come `UTF-8` , `US-ASCII` , `ASCII-8BIT` ) che specifica come interpretare quei byte come caratteri.

Le stringhe di Ruby possono essere utilizzate per contenere il testo (in pratica una sequenza di caratteri), nel qual caso viene utilizzata solitamente la codifica UTF-8.

```
"abc".bytes  # => [97, 98, 99]
"abc".encoding.name  # => "UTF-8"
```

Le stringhe di Ruby possono anche essere utilizzate per contenere dati binari (una sequenza di byte), nel qual caso viene utilizzata solitamente la codifica ASCII-8BIT.

```
[42].pack("i").encoding  # => "ASCII-8BIT"
```

È possibile che la sequenza di byte in una stringa non corrisponda alla codifica, causando errori se si tenta di utilizzare la stringa.

```
"\xFF \xFF".valid_encoding? # => false
"\xFF \xFF".split(' ')      # ArgumentError: invalid byte sequence in UTF-8
```

## Sostituzione delle stringhe

```
p "This is %s" % "foo"
# => "This is foo"

p "%s %s %s" % ["foo", "bar", "baz"]
# => "foo bar baz"

p "%{foo} == %{foo}" % {:foo => "foo" }
# => "foo == foo"
```

Vedi String `%` docs e Kernel :: sprintf per maggiori dettagli.

## La stringa inizia con

Per scoprire se una stringa inizia con un pattern, il `start_with?` il metodo è utile

```
str = "zebras are cool"
str.start_with?("zebras")      => true
```

Puoi anche controllare la posizione del pattern con l' `index`

```
str = "zebras are cool"
str.index("zebras").zero?      => true
```

## La stringa finisce con

Per scoprire se una stringa termina con un pattern, `end_with?` il metodo è utile

```
str = "I like pineapples"
str.end_with?("pineaaples")      => false
```

## Posizionamento delle stringhe

In Ruby, le stringhe possono essere giustificate a sinistra, giustificate a destra o centrate

Per la stringa di giustificazione a sinistra, utilizzare il metodo `ljust` . Questo accetta due parametri, un numero intero che rappresenta il numero di caratteri della nuova stringa e una stringa, che rappresenta il modello da riempire.

Se il numero intero è maggiore della lunghezza della stringa originale, la nuova stringa verrà giustificata a sinistra con il parametro stringa opzionale che occupa lo spazio rimanente. Se il parametro stringa non viene fornito, la stringa verrà riempita di spazi.

```
str ="abcd"
str.ljust(4)           => "abcd"
str.ljust(10)          => "abcd      "
```

Per giustificare a destra una stringa, utilizzare il metodo `rjust` . Questo accetta due parametri, un numero intero che rappresenta il numero di caratteri della nuova stringa e una stringa, che rappresenta il modello da riempire.

Se il numero intero è maggiore della lunghezza della stringa originale, la nuova stringa sarà giustificata a destra con il parametro stringa opzionale che occupa lo spazio rimanente. Se il parametro stringa non viene fornito, la stringa verrà riempita di spazi.

```
str = "abcd"
str.rjust(4)           => "abcd"
str.rjust(10)          => "      abcd"
```

Per centrare una corda, usa il metodo `center` . Questo accetta due parametri, un numero intero che rappresenta la larghezza della nuova stringa e una stringa, con la quale verrà riempita la stringa originale. La stringa sarà allineata al centro.

```
str = "abcd"
str.center(4)           => "abcd"
str.center(10)          => "   abcd   "
```

Leggi stringhe online: https://riptutorial.com/it/ruby/topic/834/stringhe

# Capitolo 66: struct

## Sintassi

- Struttura = Struct.new: attributo

## Examples

### Creare nuove strutture per i dati

`Struct` definisce nuove classi con gli attributi specificati e i metodi di accesso.

```
Person = Struct.new :first_name, :last_name
```

È quindi possibile creare un'istanza degli oggetti e utilizzarli:

```
person = Person.new 'John', 'Doe'
# => #<struct Person first_name="John", last_name="Doe">

person.first_name
# => "John"

person.last_name
# => "Doe"
```

### Personalizzazione di una classe di struttura

```
Person = Struct.new :name do
  def greet(someone)
    "Hello #{someone}! I am #{name}!"
  end
end

Person.new('Alice').greet 'Bob'
# => "Hello Bob! I am Alice!"
```

### Ricerca degli attributi

È possibile accedere agli attributi stringhe e simboli come chiavi. Anche gli indici numerici funzionano.

```
Person = Struct.new :name
alice = Person.new 'Alice'

alice['name']  # => "Alice"
alice[:name]   # => "Alice"
alice[0]       # => "Alice"
```

Leggi struct online: https://riptutorial.com/it/ruby/topic/5016/struct

# Capitolo 67: Tempo

## Sintassi

- `Time.now`
- `Time.new([year], [month], [day], [hour], [min], [sec], [utc_offset])`

## Examples

### Come utilizzare il metodo strftime

Convertire una volta in una stringa è una cosa abbastanza comune da fare in Ruby. `strftime` è il metodo che si usa per convertire il tempo in una stringa.

Ecco alcuni esempi:

```
Time.now.strftime("%Y-%m-d %H:%M:S") #=> "2016-07-27 08:45:42"
```

Questo può essere ulteriormente semplificato

```
Time.now.strftime("%F %X")  #=> "2016-07-27 08:45:42"
```

### Creare oggetti temporali

Ottieni l'ora corrente:

```
Time.now
Time.new # is equivalent if used with no parameters
```

Ottieni un tempo specifico:

```
Time.new(2010, 3, 10) #10 March 2010 (Midnight)
Time.new(2015, 5, 3, 10, 14) #10:14 AM on 3 May 2015
Time.new(2050, "May", 3, 21, 8, 16, "+10:00") #09:08:16 PM on 3 May 2050
```

Per convertire un tempo in un'epoca puoi usare il metodo `to_i` :

```
Time.now.to_i # => 1478633386
```

Puoi anche convertire da epoch a Time usando il metodo `at` :

```
Time.at(1478633386) # => 2016-11-08 17:29:46 -0200
```

Leggi Tempo online: https://riptutorial.com/it/ruby/topic/4346/tempo

# Capitolo 68: Test dell'API RSPec JSON puro

## Examples

### Testare l'oggetto Serializer e introdurlo su Controller

Diciamo che vuoi costruire la tua API per rispettare le specifiche di jsonapi.org e il risultato dovrebbe essere simile a:

```
{
  "article": {
    "id": "305",
    "type": "articles",
    "attributes": {
      "title": "Asking Alexandria"
    }
  }
}
```

Test per l'oggetto Serializer potrebbe assomigliare a questo:

```
# spec/serializers/article_serializer_spec.rb

require 'rails_helper'

RSpec.describe ArticleSerializer do
  subject { described_class.new(article) }
  let(:article) { instance_double(Article, id: 678, title: "Bring Me The Horizon") }

  describe "#as_json" do
    let(:result) { subject.as_json }

    it 'root should be article Hash' do
      expect(result).to match({
        article: be_kind_of(Hash)
      })
    end

    context 'article hash' do
      let(:article_hash) { result.fetch(:article) }

      it 'should contain type and id' do
        expect(article_hash).to match({
          id: article.id.to_s,
          type: 'articles',
          attributes: be_kind_of(Hash)
        })
      end

      context 'attributes' do
        let(:article_hash_attributes) { article_hash.fetch(:attributes) }

        it do
          expect(article_hash_attributes).to match({
            title: /[Hh]orizon/,
```

```
          })
        end
      end
    end
  end
end
```

L'oggetto Serializer può avere questo aspetto:

```
# app/serializers/article_serializer.rb

class ArticleSerializer
  attr_reader :article

  def initialize(article)
    @article = article
  end

  def as_json
    {
      article: {
        id: article.id.to_s,
        type: 'articles',
        attributes: {
          title: article.title
        }
      }
    }
  end
end
```

Quando eseguiamo le nostre specifiche "serializzatori" tutto passa.

È piuttosto noioso. Introduciamo un errore di battitura nel nostro serializzatore di articoli: invece di `type: "articles"` restituiamo il `type: "events"` ed eseguiamo nuovamente i nostri test.

```
rspec spec/serializers/article_serializer_spec.rb

.F.

Failures:

  1) ArticleSerializer#as_json article hash should contain type and id
     Failure/Error:
       expect(article_hash).to match({
         id: article.id.to_s,
         type: 'articles',
         attributes: be_kind_of(Hash)
       })

       expected {:id=>"678", :type=>"event",
:attributes=>{:title=>"Bring Me The Horizon"}} to match {:id=>"678",
:type=>"articles", :attributes=>(be a kind of Hash)}
       Diff:

       @@ -1,4 +1,4 @@
       -:attributes => (be a kind of Hash),
       +:attributes => {:title=>"Bring Me The Horizon"},
```

```
        :id => "678",
       -:type => "articles",
       +:type => "events",

     # ./spec/serializers/article_serializer_spec.rb:20:in `block (4
levels) in <top (required)>'
```

Una volta eseguito il test, è facile individuare l'errore.

Una volta che hai corretto l'errore (correggi il tipo di `article` ), puoi introdurlo su Controller in questo modo:

```
# app/controllers/v2/articles_controller.rb
module V2
  class ArticlesController < ApplicationController
    def show
      render json: serializer.as_json
    end

    private
      def article
        @article ||= Article.find(params[:id])
      end

      def serializer
        @serializer ||= ArticleSerializer.new(article)
      end
  end
end
```

Questo esempio si basa sull'articolo: http://www.eq8.eu/blogs/30-pure-rspec-json-api-testing

Leggi Test dell'API RSPec JSON puro online: https://riptutorial.com/it/ruby/topic/7842/test-dell-api-rspec-json-puro

# Capitolo 69: truthiness

## Osservazioni

Come regola generale, evitare l'uso di doppie negazioni nel codice. Rubocop dice che le doppie negazioni sono inutilmente complesse e possono spesso essere sostituite con qualcosa di più leggibile.

Invece di scrivere

```
def user_exists?
    !!user
end
```

uso

```
def user_exists?
    !user.nil?
end
```

## Examples

### Tutti gli oggetti possono essere convertiti in booleani in Ruby

Utilizzare la sintassi di doppia negazione per verificare la veridicità dei valori. Tutti i valori corrispondono a un valore booleano, indipendentemente dal loro tipo.

```
irb(main):001:0> !!1234
=> true
irb(main):002:0> !!"Hello, world!"
(irb):2: warning: string literal in condition
=> true
irb(main):003:0> !!true
=> true
irb(main):005:0> !!{a:'b'}
=> true
```

Tutti i valori tranne `nil` e `false` sono veri.

```
irb(main):006:0> !!nil
=> false
irb(main):007:0> !!false
=> false
```

### La verità di un valore può essere usata nei costrutti if-else

Non è necessario utilizzare la doppia negazione nelle istruzioni if-else.

```
if 'hello'
    puts 'hey!'
else
    puts 'bye!'
end
```

Il codice sopra stampa 'hey!' sullo schermo.

Leggi truthiness online: https://riptutorial.com/it/ruby/topic/5852/truthiness

# Capitolo 70: Uso della gemma

## Examples

### Installare gemme di rubini

Questa guida presuppone che tu abbia già installato Ruby. Se stai usando Ruby < `1.9` dovrai installare manualmente RubyGems in quanto non sarà incluso in modo nativo .

Per installare una gemma ruby, inserisci il comando:

```
gem install [gemname]
```

Se stai lavorando su un progetto con un elenco di dipendenze gem, queste saranno elencate in un file chiamato `Gemfile` . Per installare una nuova gemma nel progetto, aggiungi la seguente riga di codice nel `Gemfile` :

```
gem 'gemname'
```

Questo `Gemfile` è usato dalla gemma Bundler per installare le dipendenze richieste dal tuo progetto, ma ciò significa che dovrai prima installare Bundler eseguendo (se non lo hai già fatto):

```
gem install bundler
```

Salvare il file e quindi eseguire il comando:

```
bundle install
```

# Specifica delle versioni

Il numero di versione può essere specificato sul comando live, con il flag `-v` , ad esempio:

```
gem install gemname -v 3.14
```

Quando si specificano i numeri di versione in un `Gemfile` , sono disponibili diverse opzioni:

- Nessuna versione specificata ( `gem 'gemname')` - Installa l' *ultima* versione compatibile con altre gemme nel `Gemfile` .
- Versione esatta specificata ( `gem 'gemname', '3.14'` ): tenterà solo di installare la versione `3.14` (e fallirà se questo non è compatibile con altre gemme nel `Gemfile` ).
- Numero minimo di versione **ottimistico** ( `gem 'gemname', '>=3.14'` ) - `gem 'gemname', '>=3.14'` solo di installare l' *ultima* versione compatibile con altre gemme nel `Gemfile` e fallirà se nessuna versione superiore o uguale a `3.14` è compatibile. L'operatore `>` può anche essere usato.

- Numero di versione minimo **pessimistico** ( `gem 'gemname', '~>3.14'` ) - Questo è funzionalmente equivalente all'uso di `gem 'gemname', '>=3.14', '<4'` . In altre parole, è consentito aumentare solo il numero dopo il *periodo finale* .

---

**Come procedura:** Si potrebbe desiderare di utilizzare una delle librerie di gestione versione di Ruby come rbenv o rvm . Attraverso queste librerie, è possibile installare di conseguenza diverse versioni di runtime e gemme di Ruby. Quindi, quando si lavora in un progetto, questo sarà particolarmente utile perché la maggior parte dei progetti è codificata contro una versione nota di Ruby.

## Installazione gemma da github / filesystem

Puoi installare una gemma da github o dal filesystem. Se la gemma è stata estratta da git o in qualche modo già sul file system, puoi installarla usando

```
gem install --local path_to_gem/filename.gem
```

Installare gem da github. Scarica i sorgenti da github

```
mkdir newgem
cd newgem
git clone https://urltogem.git
```

Costruisci la gemma

```
gem build GEMNAME.gemspec
gem install gemname-version.gem
```

## Verifica se una gemma richiesta è installata dal codice

Per verificare se una gemma richiesta è installata, dal tuo codice, puoi usare quanto segue (usando nokogiri come esempio):

```
begin
  found_gem = Gem::Specification.find_by_name('nokogiri')
  require 'nokogiri'
  ....
  <the rest of your code>
rescue Gem::LoadError
end
```

Tuttavia, questo può essere ulteriormente esteso a una funzione che può essere utilizzata nella configurazione della funzionalità all'interno del codice.

```
def gem_installed?(gem_name)
  found_gem = false
  begin
    found_gem = Gem::Specification.find_by_name(gem_name)
  rescue Gem::LoadError
```

```
    return false
  else
    return true
  end
end
```

Ora puoi controllare se è installato il gem richiesto e stampare un messaggio di errore.

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  printf "nokogiri gem required\n"
  exit 1
end
```

o

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  require 'REXML'
end
```

## Usando un Gemfile e Bundler

Un `Gemfile` è il modo standard per organizzare le dipendenze nell'applicazione. Un Gemfile di base sarà simile a questo:

```
source 'https://rubygems.org'

gem 'rack'
gem 'sinatra'
gem 'uglifier'
```

Puoi specificare le versioni della gemma che desideri come segue:

```
# Match except on point release. Use only 1.5.X
gem 'rack', '~>1.5.2'
# Use a specific version.
gem 'sinatra', '1.4.7'
# Use at least a version or anything greater.
gem 'uglifier', '>= 1.3.0'
```

Puoi anche estrarre gemme direttamente da un repository git:

```
# pull a gem from github
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git'
# you can specify a sha
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', sha:
'30d4fb468fd1d6373f82127d845b153f17b54c51'
# you can also specify a branch, though this is often unsafe
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', branch: 'master'
```

Puoi anche raggruppare le gemme a seconda di cosa sono usate. Per esempio:

```
group :development, :test do
    # This gem is only available in dev and test, not production.
    gem 'byebug'
end
```

È possibile specificare su quale piattaforma eseguire determinate gem se l'applicazione deve essere in grado di essere eseguita su più piattaforme. Per esempio:

```
platform :jruby do
  gem 'activerecord-jdbc-adapter'
  gem 'jdbc-postgres'
end

platform :ruby do
  gem 'pg'
end
```

Per installare tutte le gemme da un Gemfile fai:

```
gem install bundler
bundle install
```

## Bundler / inline (bundler v1.10 e successive)

A volte è necessario creare una sceneggiatura per qualcuno ma non si è sicuri di cosa abbia sulla sua macchina. C'è tutto ciò di cui ha bisogno il tuo script? Da non preoccuparsi. Bundler ha una grande funzione chiamata in linea.

Fornisce un metodo `gemfile` e prima `gemfile` dello script viene scaricato e richiede tutte le gemme necessarie. Un piccolo esempio:

```
require 'bundler/inline' #require only what you need

#Start the bundler and in it use the syntax you are already familiar with
gemfile(true) do
  source 'https://rubygems.org'
        gem 'nokogiri', '~> 1.6.8.1'
        gem 'ruby-graphviz'
end
```

Leggi Uso della gemma online: https://riptutorial.com/it/ruby/topic/1540/uso-della-gemma

# Capitolo 71: Valutazione dinamica

## Sintassi

- eval "fonte"
- eval "fonte", vincolante
- eval "fonte", proc
- binding.eval "source" # uguale a `eval "source", binding`

## Parametri

| Parametro | Dettagli |
|---|---|
| `"source"` | Qualsiasi codice sorgente di Ruby |
| `binding` | Un'istanza della classe `Binding` |
| `proc` | Un'istanza di classe `Proc` |

## Examples

### Valutazione delle istanze

Il metodo `instance_eval` è disponibile su tutti gli oggetti. Valuta il codice nel contesto del destinatario:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable  # => :value
```

`instance_eval` imposta `self` per `object` per la durata del blocco di codice:

```
object.instance_eval { self == object }  # => true
```

Il destinatario viene anche passato al blocco come unico argomento:

```
object.instance_eval { |argument| argument == object }  # => true
```

Il metodo `instance_exec` differisce in questo senso: passa invece i suoi argomenti al blocco.

```
object.instance_exec :@variable do |name|
```

```
  instance_variable_get name  # => :value
end
```

## Valutare una stringa

Qualsiasi `String` può essere valutata in fase di esecuzione.

```
class Example
  def self.foo
    :foo
  end
end

eval "Example.foo" #=> :foo
```

## Valutare all'interno di un legame

Ruby tiene traccia delle variabili locali e della variabile `self` tramite un oggetto chiamato binding. Possiamo ottenere l'associazione di un ambito con il richiamo del `Kernel#binding` e valutare la stringa all'interno di un'associazione tramite `Binding#eval` .

```
b = proc do
  local_variable = :local
  binding
end.call

b.eval "local_variable" #=> :local
```

---

```
def fake_class_eval klass, source = nil, &block
  class_binding = klass.send :eval, "binding"

  if block
    class_binding.local_variable_set :_fake_class_eval_block, block
    class_binding.eval "_fake_class_eval_block.call"
  else
    class_binding.eval source
  end
end

class Example
end

fake_class_eval Example, <<-BLOCK
  def self.foo
    :foo
  end
BLOCK

fake_class_eval Example do
  def bar
    :bar
  end
end

Example.foo #=> :foo
```

```
Example.new.bar #=> :bar
```

## Creare dinamicamente i metodi dalle stringhe

Ruby offre `define_method` come metodo privato su moduli e classi per la definizione di nuovi metodi di istanza. Tuttavia, il "corpo" del metodo deve essere un `Proc` o un altro metodo esistente.

Un modo per creare un metodo dai dati di stringa grezzi è utilizzare `eval` per creare un Proc dal codice:

```
xml = <<ENDXML
<methods>
  <method name="go">puts "I'm going!"</method>
  <method name="stop">7*6</method>
</methods>
ENDXML

class Foo
  def self.add_method(name,code)
    body = eval( "Proc.new{ #{code} }" )
    define_method(name,body)
  end
end

require 'nokogiri' # gem install nokogiri
doc = Nokogiri.XML(xml)
doc.xpath('//method').each do |meth|
  Foo.add_method( meth['name'], meth.text )
end

f = Foo.new
p Foo.instance_methods(false)  #=> [:go, :stop]
p f.public_methods(false)      #=> [:go, :stop]
f.go                           #=> "I'm going!"
p f.stop                       #=> 42
```

Leggi Valutazione dinamica online: https://riptutorial.com/it/ruby/topic/5048/valutazione-dinamica

# Capitolo 72: variabili ambientali

## Sintassi

- ENV [nome_variabile]
- ENV.fetch (nome_variabile, valore_predefinito)

## Osservazioni

Consenti di ottenere il percorso del profilo utente in modo dinamico per lo scripting in Windows

## Examples

### Esempio per ottenere il percorso del profilo utente

```
# will retrieve my home path
ENV['HOME'] # => "/Users/username"

# will try retrieve the 'FOO' environment variable. If failed, will get 'bar'
ENV.fetch('FOO', 'bar')
```

Leggi variabili ambientali online: https://riptutorial.com/it/ruby/topic/4276/variabili-ambientali

# Titoli di coda

| S. No | Capitoli | Contributors |
|---|---|---|
| 1 | Iniziare con Ruby Language | alejosocorro, CalmBit, Community, ctietze, Darpan Chhatravala , David Grayson, DawnPaladin, Eli Sadoff, Jonathan_W, Jonathon Jones, Ken Y-N, knut, Lucas Costa, luissimo, Martin Velez, Mhmd, mnoronha, numbermaniac, peter, prcastro, RamenChef, Simone Carletti, smileart, Steve, Timo Schilling, Tom Lord, Tot Zam, Undo, Vishnu Y S, Wayne Conrad |
| 2 | Ambito e visibilità variabili | Matheus Moreira, Ninigi, Sandeep Tuniki |
| 3 | Applicazioni a riga di comando | Eli Sadoff |
| 4 | Appuntamento | Austin Vern Songer, Redithion |
| 5 | Argomenti della parola chiave | giniouxe, mnoronha, Simone Carletti |
| 6 | Array | Ajedi32, alebruck, Andrea Mazzarella, Andrey Deineko, Automatico, br3nt, Community, Dalton, daniero, David Grayson, davidhu2000, DawnPaladin, D-side, Eli Sadoff, Francesco Lupo Renzi, iGbanam, joshaidan, Katsuhiko Yoshida, knut, Lucas Costa, Lukas Baliak, lwassink, Masa Sakano, meagar, Mhmd, Mike H-R, MrTheWalrus, ndn, Nick Roz, nus, Pablo Torrecilla, Pooyan Khosravi, Richard Hamilton, Sagar Pandya, Saroj Sasmal, Shadoath, squadette, Steve, Tom Lord, Undo, Vasfed |
| 7 | Array multidimensionali | Francesco Boffa |
| 8 | Blocchi e Procs e Lambdas | br3nt, coreyward, Eli Sadoff, engineersmnky, Jasper, Kathryn, Lukas Baliak, Marc-Andre, Matheus Moreira, meagar, Mhmd, nus, Pooyan Khosravi, QPaysTaxes, Simone Carletti |
| 9 | C estensioni | Austin Vern Songer, photoionized |
| 10 | Caricamento dei file di origine | mnoronha, nus |
| 11 | Casting (conversione del tipo) | giniouxe, Jon Wood, meagar, Mhmd, Nakilon |

| | | |
|---|---|---|
| 12 | Cattura le eccezioni con Begin / Rescue | Sean Redmond, stevendaniels |
| 13 | Classi | br3nt, davidhu2000, Elenian, Eric Bouchut, giniouxe, JoeyB, Jon Wood, Justin Chadwell, Lukas Baliak, Martin Velez, MegaTom, Mhmd, Nick Roz, nus, philomory, Simone Carletti, spencer.sm, stevendaniels, thesecretmaster |
| 14 | Coda | Pooyan Khosravi |
| 15 | Commenti | giniouxe, Jeremy, Rahul Singh, Robert Harvey |
| 16 | costanti | Engr. Hasanuzzaman Sumon, mahatmanich, user2367593 |
| 17 | Costanti speciali in Ruby | giniouxe, mnoronha, Redithion |
| 18 | Creazione / gestione gemma | manasouza, thesecretmaster |
| 19 | Debug | DawnPaladin, ogirginc |
| 20 | Design Patterns e Idioms in Ruby | 4444, alexunger, Ali MasudianPour, Divya Sharma, djaszczurowski, Lucas Costa, user1213904 |
| 21 | destrutturazione | Austin Vern Songer, Zaz |
| 22 | eccezioni | David Grayson, Eric Bouchut, hillary.fraley, iturgeon, kamaradclimber, Lomefin, Lucas Costa, Lukas Baliak, lwassink, Michael Kuhinica, moertel, Muhammad Abdullah, ndn, Robert Columbia, Simone Carletti, Steve, Vasfed, Wayne Conrad |
| 23 | Enumerabile in Ruby | Neha Chopra |
| 24 | enumeratori | errm, Matheus Moreira |
| 25 | ERB | amingilani |
| 26 | Eredità | br3nt, Gaelan, Kirti Thorat, Lynn, MegaTom, mlabarca, nus, Pascal Fabig, Pragash, RamenChef, Simone Carletti, thesecretmaster, Vasfed |
| 27 | Espressioni regolari e operazioni basate su Regex | Addison, Elenian, giniouxe, Jon Ericson, moertel, mudasobwa, Nick Roz, peter, Redithion, Saša Zejnilović, Scudelletti, Shelvacu |
| 28 | Filo | Austin Vern Songer, Maxim Fedotov, MegaTom, moertel, Simone Carletti, Surya |
| 29 | Flusso di controllo | alebruck, angelparras, br3nt, daniero, DarKy, David Grayson, |

| | | |
|---|---|---|
| | | dgilperez, Dimitry_N, D-side, Elenian, Francesco Lupo Renzi, giniouxe, JoeyB, jose_castro_arnaud, kannix, Kathryn, Lahiru, mahatmanich, meagar, MegaTom, Michael Gaskill, moertel, mudasobwa, Muhammad Abdullah, ndn, Nick Roz, Pablo Torrecilla, russt, Scudelletti, Simone Carletti, Steve, the Tin Man, theIV, Tom Lord, Vasfed, Ven, vgoff, Yule |
| 30 | Gamma | DawnPaladin, Rahul Singh, Yonatha Almeida |
| 31 | Genera un numero casuale | user1821961 |
| 32 | hash | 4444, Adam Sanderson, Arman Jon Villalobos, Atul Khanduri, Bo Jeanes, br3nt, C dot StrifeVII, Charlie Egan, Charlie Harding , Christoph Petschnig, Christopher Oezbek, Community, danielrsmith, David Grayson, dgilperez, divyum, Felix, G. Allen Morris III, gorn, iltempo, Ivan, Jeweller, jose_castro_arnaud, kabuko, Kathryn, kleaver, Konstantin Gredeskoul, Koraktor, Kris, Lucas Costa, Lukas Baliak, Marc-Andre, Martin Samami, Martin Velez, Matt, MattD, meagar, MegaTom, Mhmd, Michael Kuhinica, moertel, mrlee, MZaragoza, ndn, neontapir, New Alexandria, Nic Nilov, Nick Roz, nus, Old Pro, Owen, peter50216, pjam, PJSCopeland, Pooyan Khosravi, RamenChef, Richard Hamilton, Sid, Simone Carletti, spejamchr , spickermann, Steve, stevendaniels, the Tin Man, Tom Lord, Ven, wirefox, Zaz |
| 33 | Iniziare con Hanami | Mauricio Junior |
| 34 | Installazione | Kathryn, Saša Zejnilović |
| 35 | instance_eval | Matheus Moreira |
| 36 | Introspezione | Felix, giniouxe, Justin Chadwell, MegaTom, mnoronha, Phrogz |
| 37 | Introspezione in Ruby | Engr. Hasanuzzaman Sumon, suhao399 |
| 38 | IRB | David Grayson, Maxim Fedotov, Saša Zejnilović |
| 39 | Iterazione | Charan Kumar Borra, Chris, Eli Sadoff, giniouxe, JCorcuera, Maxim Pontyushenko, MegaTom, ndn, Nick Roz, Ozgur Akyazi, Qstreet, SajithP, Simone Carletti |
| 40 | JSON con Ruby | Alu |
| 41 | Messaggio in corso | Pooyan Khosravi |
| 42 | metaprogrammazione | C dot StrifeVII, giniouxe, Matheus Moreira, MegaTom, meta, Phrogz, Pooyan Khosravi, Simon Soriano, Sourabh Upadhyay |

| | | |
|---|---|---|
| 43 | method_missing | Adam Sanderson, Artur Tsuda, mnoronha, Nick Roz, Tom Harrison Jr, Yule |
| 44 | metodi | Adam Sanderson, Artur Tsuda, br3nt, David Ljung Madison, fairchild, giniouxe, Kathryn, mahatmanich, Nick Podratz, Nick Roz, nus, Redithion, Simone Carletti, Szymon Włochowski, Thomas Gerot, Zaz |
| 45 | Modificatori di accesso rubino | Neha Chopra |
| 46 | moduli | giniouxe, Lynn, MegaTom, mrcasals, nus, RamenChef, Vasfed |
| 47 | Monkey Patching in Ruby | Dorian, paradoja, RamenChef |
| 48 | Numeri | alexunger, Eli Sadoff, ndn, Redithion, Richard Hamilton, Simone Carletti, Steve, Tom Lord, wirefox |
| 49 | Operatore Splat (*) | Kathryn |
| 50 | operatori | ArtOfCode, Jonathan, nus, Phrogz, Tom Harrison Jr |
| 51 | Operazioni su file e I / O | Doodad, KARASZI István, Martin Velez, max pleaner, Milo P, mnoronha, Nuno Silva, thesecretmaster |
| 52 | OptionParser | Kathryn |
| 53 | paragonabile | giniouxe, ndn, sandstrom, sonna |
| 54 | perfezionamenti | max pleaner, xavdid |
| 55 | rbenv | Kathryn, Vidur |
| 56 | Ricevitori impliciti e Sé comprensivo | Andrew |
| 57 | Ricorsione in Ruby | jphager2, Kathryn, SajithP |
| 58 | Ruby Version Manager | Alu, giniouxe, Hardik Kanjariya ツ |
| 59 | simboli | Artur Tsuda, Arun Kumar M, Nick Podratz, Owen, pjrebsch, Pooyan Khosravi, Simone Carletti, Tom Lord, walid |
| 60 | Singleton Class | Geoffroy, giniouxe, Matheus Moreira, MegaTom, nus, Pooyan Khosravi |
| 61 | Sistema operativo o comandi Shell | Roan Fourie |

| | | |
|---|---|---|
| 62 | stringhe | AJ Gregory, br3nt, Charlie Egan, Community, David Grayson, davidhu2000, Jon Ericson, Julian Kohlman, Kathryn, Lucas Costa, Lukas Baliak, meagar, Muhammad Abdullah, NateW, Nick Roz, Phil Ross, Richard Hamilton, sandstrom, Sid, Simone Carletti, Steve, Vasfed, Velocibadgery, wjordan |
| 63 | struct | Matheus Moreira |
| 64 | Tempo | giniouxe, Lucas Costa, MegaTom, stevendaniels |
| 65 | Test dell'API RSPec JSON puro | equivalent8, RamenChef |
| 66 | truthiness | giniouxe, Umang Raghuvanshi |
| 67 | Uso della gemma | Anthony Staunton, Brian, Inanc Gumus, mnoronha, MZaragoza , NateSHolland, Saša Zejnilović, SidOfc, Simone Carletti, thesecretmaster, Tom Lord, user1489580 |
| 68 | Valutazione dinamica | Matheus Moreira, MegaTom, Phrogz, Pooyan Khosravi, Simone Carletti |
| 69 | variabili ambientali | Lucas Costa, mnoronha, snonov |