RUBY vers. 5

Aggiornato il giorno 22 Maggio 2011

Un ringraziamento a Giulio Ardoino per il contributo.

```
#  Lo si mette prima di ogni commento
=begin
  Questo è
  un commento
  su più righe
=end
```

Le stringhe si scrivono tra ‘ ’ oppure tra “ ” (ma in questo caso verranno interpretate) ad es.:

```
a = 2   (i numeri vanno scritti senza gli apici singoli o doppi)
puts 'a'    # stamperà a    perché tra le parentesi la variabile a è interpretata come stringa
puts "a"   # stamperà a    perché tra le parentesi la variabile a è interpretata come stringa
puts a      # stamperà 2   perché la variabile a è un numero


a = ‘casa’    (le stringhe vanno scritte con gli apici altrimenti avremo un errore)
puts 'a'    #  stamperà a
puts "a"  #  stamperà a
puts a      # stamperà casa


Un dato si stampa a video con l'istruzione puts   (la s sta per stringa)
puts 4 + 4                  #--> stamperà  --  8
puts 5**2                   #--> stamperà  --  25 (5 al quadrato)
puts ' '                    #--> sarà vuoto
puts 'You'are back'         #--> darà errore
puts 'You\'re back'         #--> stamperà  --  You're back
+ unisce le stringhe e/o i numeri
```

Le variabili

Le variabili non sono oggetti.
Dichiarare una Variabile:

```
variabileuno = 'casa'
variabiledue = 2            # --->(i numeri vanno scritti senza gli apici, a meno che non vogliamo
che 2 sia una stringa)
puts variabileuno *2        #--> stamperà  --  casacasa
variabiletre = ' casa'
```

```
puts variabiletre * 2        #--> stamperà  --  casa casa
var1 = 20
var2 = var1
puts var1                    #--> stamperà  --  20
puts var 2                   #--> stamperà  --  20
var1 = 'seven'
puts var1                    #-->stamperà  --  seven
puts var2                    #-->stamperà  --  20
```

I Metodi

I metodi convertono i dati delle variabili
variabile.convertitore
.to_s --> converte la variabile in stringa
.to_i --> converte la variabile in numero

```
var1 = 5
var2 = '12'
puts var1.to_s + var2  #--> stamperà  --  512 (come stringa e non come numero)
var1 = 5
var2 = '12'
puts var1 + var2.to_i   #--> stamperà  --  17 (come numero e non come stringa)
```

Ottenere dati da input di tastiera

gets

```
puts 'Qual\' e\' il tuo nome ?'
nome = gets # la funzione gets apetta l'inserimento di dati da tastiera (es. Marco), ma lascia uno spazio
puts 'Ciao ' + nome + 'e\' bel nome'   #stamperà Ciao Marco (su una riga)
                                       #è bel nome (nella seconda riga)
```

gets.chomp

```
puts 'Qual\' e\' il tuo nome ?'
nome = gets.chomp # la funzione gets.chomp apetta l'inserimento di dati da tastiera (es. Marco), ma non lascia uno spazio
puts 'Ciao ' + nome + ' e\' bel nome'  #stamperà Ciao Marco è bel nome (tutto su una riga)
```

Metodi o (funzioni)

Se gli oggetti (come le stringhe, interi e virgola mobile) sono nomi in Ruby, allora i metodi sono come i verbi. Come ogni verbo ha bisogno di un nome, ogni metodo ha bisogno di un oggetto.

Ecco alcuni metodi:
.reverse (viene scritta la variabile al contrario)
.length (scrive in numeri interi "e non in stringa" la lunghezza della variabile)
.to_s (converte la variabile in stringa)
.to_i (converte la variabile in intero)
.upcase (scrive la variabile in maiuscolo)
.downcase (scrive la variabile in minuscolo)
.swapcase ( )
.capitalize (scrive l'iniziale con la lettera maiuscola)
.center (inserisce la variabile al centro)
.ljust (inserisce la variabile a sinistra)
.rjust (inserisce la variabile a destra)
.times (ripete la variabile più volte -->vedi do)
.join (unisce ---> vedi gli Array)
.slice (preleva carattere per carattere di una stringa)

Random

```
puts rand  #(visualizza numeri random con la virgola)
puts rand(1000)  #(visualizza numeri interi da 0 a 999)
puts
```

Controllo di flusso

> maggiore
< minore
== uguale a
!= diverso uguale
>= minore uguale
<= maggiore uguale
|| o
&& e

if = se
elsif = se
else = altrimenti
while = finchè
unless = a meno che
end = termina

IF ELSIF ELSE END (end va sempre inserito alla fine di ogni if e while)

```
a = 10
b = 20
if a > b
  puts 'a è maggiore di b'
elsif a == b
  puts 'a è uguale a b'
else
  puts 'a è minore di b'
end
```

WHILE END traducete while con la parola finchè (end va sempre inserito alla fine di ogni if e while)

```
puts 'Indovina il numero'
a = gets.chomp  il numero inserito verrà considerato come una stringa e non come un numero
while a != '100'                                  # finchè a è diverso da 100
  puts 'Hai inserito il numero sbagliato'   # scrivi Hai inserito il numero sbagliato
  a = gets.chomp                                  # il terminale aspetta l'inserimento di un numero
end
puts 'Hai indovinato il numero'
```

Array e Itaratori

```
              0          1          2         3
citta = ['Lucca', 'Firenze', 'Pistoia', 'Siena']
puts citta      # stamperà  --  tutte le città dall'alto verso il basso
puts ' '          # stampa una riga vuota
puts citta[0] # stamperà   --  Lucca
puts citta[2] # stamperà   --  Pistoia
puts citta[4] # stamperà   --  nil ossia niente poichè l'array 4 non esiste
```

each è un metodo apposito per gli array, va sempre usato insieme a **do** ed **end**.

do non è un metodo.

```
              0          1          2         3
citta = ['Lucca', 'Firenze', 'Pistoia', 'Siena']
citta.each do |italia|
puts 'Io ho abitato in queste citta ' + italia + '!'   # stamperà -- tutte le città dall'alto verso il basso
end
```

do, vediamo un altro uso di do (do non è un metodo)

```
do =  esegui  # da tradurre come esegui
```

```
3.times do
puts 'ciao'   #  stamperà  --  3 volte ciao dall'alto verso il basso
end

                0          1            2          3
citta = ['Lucca', 'Firenze', 'Pistoia', 'Siena']
50.times do
puts citta  # stamperà  --  50 volte Lucca Firenze Pistoia Siena dall'alto verso il basso
puts [ ]     # stamperà  --  50 volte nulla, ossia non farà nulla.
end
```

Metodi da usarsi per movimentare gli array

.push (inserisce)
.pop (rimuove l'ultimo oggetto dall'array)
.length (dice da quanti elementi in numero è composto l'array)

```
citta=[ ]
citta.push 'Lucca'    #  inserisce Lucca nell'array
puts citta.length     #  l'array è composto da 1 oggetto
citta.push 'Firenze'  #  inserisce Firenze nell'array
citta.push 'Pistoia'  #  inserisce Pistoia nell'array
puts citta.length     #  l'array è composto da 3 oggetti
citta.pop             #  cancella Pistoia, ossia l'ultimo oggetto, dall'array
puts citta.length     #  l'array è composto da 2 oggetti
```

Impariamo a scrivere i nostri metodi o funzioni.

Una funzione inizia con **def** e termina con **end**. Se gli oggetti in ruby sono come i nomi in inglese, i metodi sono come i verbi.
I metodi NON devono iniziare con la lettera maiuscola.
Questo sotto è un esempio di un metodo.

```
def prova
  puts 'ciao'
end
prova  #  stamperà --  ciao
```

Le variabili locali vivono solo dentro il metodo, se usate fuori daranno errore.
Le variabili locali (dentro il metodo) e fuori da questo, possono avere lo stesso nome, ma non interferiranno mai l'una con l'altra.
Usiamo 1 variabile locale ed 1 parametro (grazie alla variabile locale numero):

```
def prova numero
  puts 'ciao' * numero
end
prova 3  #  stamperà  --  ciao 3 volte
prova     #  darà errore perché manca un parametro (wrong number of arguments (0 for 1)
(ArgumentError)
```

Usiamo 2 variabili locali "numero e doppio" e passiamo 2 parametri, raddoppia è sempre il nome del metodo.

```
def raddoppia numero
  doppio = numero*2
  puts numero.to_s+' moltiplicato due fa ' +doppio.to_s
end
raddoppia 7 # stamperà --  7 moltiplicato fa 14
```

Usiamo 1 sola variabile locale ed una espressione alla fine del metodo.

```
def raddoppia numero
  puts 'ciao'*numero
  'sono una stringa'
end
x = raddoppia 3  #  stamperà  --  ciaociaociao
puts x                #  stamperà  --  sono una stringa
```

L'ultimo valore ritornato da 1 metodo è l'ultima espressione valutata e non l'ultima stringa nel metodo. Infatti (sono una stringa) è tra apici, il che vuol dire che è una espressione.

Passiamo adesso 3 dati ad una metodo creato da noi, prima si dichiarano quante variabili si utilizzano dichiarando il metodo (si possono mettere anche tra parentesi), poi si richiama il metodo passandogli altre variabili, basta considerare il numero (in questo caso sono 3):

```
def prova a, b, c
      puts 'buon giorno ' + a + b
end
c = 'Cristian '
d = 'Massimiliano '
e = 'Roberto'
prova c, d, e    # stamperà  --  buon giorno Cristian Massimiliano Roberto
```

Altro esempio con metodo per ftp:

```
require 'net/ftp'
def f (ip, ut, pa)
    ftp = Net::FTP.new(ip)
      ftp.login(ut, pa)
      files = ftp.chdir('/directoryinventata/')
      ftp.putbinaryfile('/Aggiornamento/file.tgz', 'file.tgz', 1024)
      ftp.putbinaryfile('/Aggiornamento/file1.tgz', 'file1.tgz', 1024)
      ftp.putbinaryfile('/Aggiornamento/file.sh', 'file.sh', 1024)
    ftp.close
end

puts 'Inserisci l\' ip'
ip = gets.chomp
puts 'Inserisci utente'
utente = gets.chomp
puts 'Inserisci password'
password = gets.chomp

f ip, utente, password
```

Le Classi

Una categoria di oggetti come i cani è chiamata classe e qualche specifico oggetto appartenente a una classe è chiamata istanza della classe.
La classe inizia sempre con la lettera maiuscola.
Cane è 1 classe
L'oggetto appartenente alla classe Cane è chiamata istanza.

Prima si definiscono le caratteristiche della classe, poi si crea una istanza.

Una categoria di oggetti come il cane è chiamata classe e qualche oggetto specifico appartenente alla classe è detta istanza di quella classe.

```
class Cane        # Creazione classe Cane
  def abbaia      # Definizione metodo abbaia
    puts "Bau Bau"
  end
end
```

new: Il metodo new crea un nuovo oggetto, cioè una nuova istanza della classe.

```
pochi = Cane.new   # Abbiamo creato una nuova istanza (pochi) della classe Cane,
                   # ora pochi è uguale a Cane ed ha le proprietà (abbaia) che
                   # avevamo deciso.

pochi.abbaia       # scriverà Bau Bau
```

Fare una nuova istanza di una classe è quello che viene chiamato istanziare quella classe. Abbiamo bisogno di avere un cane prima per avere il piacere della sua conversazione; Non possiamo semplicemente chiedere alla classe Cane di abbaiare per noi.

```
Cane.abbaia        # darà errore (non possiamo usare il metodo con la classe)


(Cane.new).abbaia  # (possiamo sentire abbaiare usando new, oltrepassando il
                   # l'istanziamento della classe, ma questo sarà solo una cosa
                   # temporanea. pochi è un istanziamento della classe permanente,
                   # Cane.new sarà solo temporaneo)

Cane.new.abbaia    # esegue temporaneamente la classe
```

Ereditarietà

Tutti i gatti sono mammiferi e tutti i mammiferi sono animali. Le classi più piccole ereditano le caratteristiche dalle classi più grandi alle quali appartengono. Se tutti i mammiferi respirano, allora tutti i gatti respirano.

```
class Mammifero      # creo la classe Mammifero
  def respira        # creo il metodo respira
    puts "Il mammifero inspira ed espira"
  end
end

class Gatto<Mammifero      # il Gatto eredita il comportamento dal Mammifero.
                           # Gatto è una sottoclasse, Mammifero è una superclasse
  def miagola              # creo il metodo miagola
    puts "Il Gatto fa mao"
  end
end
```

```
tama = Gatto.new  # creo la nuova istanza tama della classe Gatto

tama.respira          # eseguo il metodo respira della classe Mammifero ereditata
tama.miagola        # eseguo il metodo miagola della classe Gatto
```

Ci saranno situazioni dove certe proprietà della superclasse non dovrebbero essere ereditate da una particolare sottoclasse. Per es. gli uccelli di solito sanno come volare, i pinguini sono una sottoclasse incapace di volare degli uccelli.

```
class Uccello       # creo la classe Uccello
  def pennuto       # creo il metodo pennuto
      puts "Sto pulendo le mie piume"
  end
  def volo          # creo il metodo volo
      puts "Sto volando"
  end
end
```

```
usel = Uccello.new   # creo la nuova istanza usel
usel.pennuto         # eseguo il metodo pennuto che scriverà Sto pulendo le mie piume
usel.volo            # eseguo il metodo volo che scriverà Sto volando
```

```
class Pinguino<Uccello     # il Pinguino eredita il comportamento dall'Uccello
  def volo                 # creo il metodo volo
    fail "Mi dispiace. Preferisco nuotare"
  end
end
```

```
o = Pinguino.new      # istanzio la classe Pinguino
o.pennuto             # eseguo il metodo pennuto che scriverà Sto pulendo le mie piume
c.volo                # eseguo il metodo volo che NON scriverà nulla essendo in fail
```

Redifinizione dei metodi

In una sottoclasse, possiamo cambiare il comportamento delle istanze ridefinendo i metodi della superclasse.

```
class Umano
  def identita
    puts "Sono una persona"
  end
```

```
  def tassatreno(eta)
    if eta < 12
      puts "Tariffa ridotta";
    else
      puts "Tariffa normale";
    end
  end
end

a = Umano.new
a.identita  (oppure Umano.new.identita)      # esegue temporaneamente la classe e stamperà a
video Sono una persona



class Studente1<Umano
  def identita
    puts "Sono uno studente"
  end
end

b = Studente.new
b.identita   (oppure Studente1.new.identita)   # esegue temporaneamente la classe e stamperà a
video Sono uno studente
```

Ora supponiamo di voler migliorare il metodo identita della superclasse e di rimpiazzarla interamente. Per questo possiamo usare la parola super.

```
class Studente2<Umano
  def identita
     super
     puts "Anche io sono uno studente"
  end
end

c = Studente2.new
```

c.identita (oppure Studente2.new.identita)   # esegue temporaneamente la classe stamperà Sono una persona e
                                # Anche io sono uno studente

Quindi super ci lascia passare argomenti al metodo originale, in questo caso (identita)

```
class Disonesto<Umano
  def tassatreno(eta)
    super(11)                          # vogliamo una tassa più bassa.
  end
end

d = Disonesto.new
d.tassatreno(25) (oppure Disonesto.new.tassatreno(25) )      # esegue temporaneamente la classe stamperà Tariffa ridotta
```

```
class Onesto<Umano
  def tassatreno(eta)
    super(eta)                         # Gli passiamo l'argomento che ci è stato dato
  end
end

e = Onesto.new
e.tassatreno(25)  (oppure Onesto.new.tassatreno(25) )   # esegue temporaneamente la classe stamperà Tariffa normale
```

Controlli di accesso

In ruby non ha funzioni, ma solo metodi. Esistono più tipi di metodi

```
def square(n)
  n * n
end

puts square(5)  # stamperà 25

class Quadrato
```

```
  def quarta_potenza_di(x)
    square(x) * square(x)                    # lasciare scritto square (quadrato), non cambiare
  end
end

Quadrato.new.quarta_potenza_di 10     # a video stamperà  10000

Non abbiamo il permesso esplicito di applicare il metodo ad un oggetto
"fish".square(5)                          # otterremo errore a video

class Test
   def due_volte(a)
     puts "#{a} due volte fa #{engine(a)}"
   end
   def engine(b)
     b*2
   end
private:engine                  # Questo nasconde engine dagli utenti this hides engine from users
end

test = Test.new

#test.engine(6)    # stamperà errore a video, engine è inaccessibile quando stiamo agendo come
                   # un utente sull'oggetto Test

test.due_volte(6)           # stamperà 6 due volte fa 12
          o
Test.new.due_volte(6)    # stamperà 6 due volte fa 12
```

Scrivere e leggere un file

Scrivere in un file:

```
file = 'prova.txt'                        # Inserisco il nome del file in una variabile
testo = 'Prova di scrittura in un file'   # Inserisco il testo
File.open file, 'w' do |f|                # Uso il metodo File.open leggo file, con il ciclo do
                                                leggo byte per byte
  f.write testo                           # Scrivo la stringa della variabile testo
end                                       # Chiudo il ciclo do
```

Leggere un file (2 possibilità):

1 possibilità:

```
file = 'prova.txt'                        # Inserisco il nome del file in una variabile
lettura = File.read file                  # Uso il metodo File.read per leggere la variabile file
puts lettura                              # Utilizzando puts leggo la variabile lettura
```

2 possibilità:

```
IO.foreach("prova.txt") { |line| puts line } # Per ciascuna linea letta da prova.txt stampa a video una linea
```

Leggere una riga di 1 file e prelevare dei caratteri:

```
arr = IO.readlines("rimmer.txt")      #  lettura del file rimmer.txt
a = arr[6]                            #  Inserisco la riga 5 dell'array nella variabile a
puts a.slice(9..14)                   #  stampo le lettere dalla 9 alla 14 della riga 5 dell'array
```

Scrittura e lettura di un file utilizzando il metodo yaml ed usando un array

```
require 'yaml'                        # Utilizzo il metodo yaml
array = ['ciao', 'io', 'sono', 'Luca']  # Inserisco dei dati in un array
test = array.to_yaml                  # Utilizzando il metodo yaml, inserisco l'array nella
                                      #  variabile testo
puts test                             # Stampiamo a video la variabile test

file = 'rimmer.txt'                   # Creiamo il file rimmer.txt
File.open file, 'w' do |f|            # Uso il metodo File.open in scrittura per scrivere
                                      #  nella variabile file con do
f.write test                          # Scrivo la variabile test (l'array) nel file
end                                   # Termino il ciclo end

lettura = File.read file      # Nella variabile lettura inserisco la lettura fatta del file rimmer.txt
leggiarray = YAML::load lettura       # Usando YAML::load leggo il file lettura che inserisco
                                      #  in una variabile
puts leggiarray                       # Leggo leggiarray
```

Installare un Modulo Esterno sotto Windows o Linux

Installare un Modulo Esterno sia sotto Windows che sotto Linux:
Scaricare e scompattare il file e poi eseguire:

```
ruby setup.rb config
```

ruby setup.rb setup
ruby setup.rb install

Eseguire il modulo ssh in uno script

```
require 'net/ssh'

Net::SSH.start('172.xxx.xxx..xxx', 'utente', :password => "123456789") do |ssh|
output = ssh.exec!("hostname")
ssh.exec "dir"
end
```

Eseguire il modulo ftp **(vedere pagina 6)**

```
require 'net/ftp'
```