Laboratorio di Elementi di Bioinformatica

Laurea Triennale in Informatica
(codice: E3101Q116)

AA 2014/2015

LEZIONE 1 - Introduzione a Ruby e tipi standard

Docente del laboratorio: Raffaella Rizzi

Introduzione a Ruby

Un codice Ruby deve essere contenuto in un file di puro testo che ha nome `*.rb`.

Per lanciarlo nella *shell* basta digitare:

```
>ruby *.rb
```

La funzione `puts`

```
puts arg1, arg2, arg3, …, argn
```

Produce in STDOUT ognuno degli argomenti passati dopo averli convertiti in stringa di caratteri, stampando dopo ogni argomento un carattere di *newline*. Qualche esempio:

La funzione `puts`

```
puts arg1, arg2, arg3, …, argn
```

Produce in STDOUT ognuno degli argomenti passati dopo averli convertiti in stringa di caratteri, stampando dopo ogni argomento un carattere di *newline*. Qualche esempio:

```
puts 4
```

```
>4
>
```

La funzione `puts`

```
puts arg1, arg2, arg3, …, argn
```

Produce in STDOUT ognuno degli argomenti passati dopo averli convertiti in stringa di caratteri, stampando dopo ogni argomento un carattere di *newline*. Qualche esempio:

```
puts 4
```

```
>4
>
```

```
puts 1+3
```

```
>4
>
```

La funzione `puts`

```
puts arg1, arg2, arg3, …, argn
```

Produce in STDOUT ognuno degli argomenti passati dopo averli convertiti in stringa di caratteri, stampando dopo ogni argomento un carattere di *newline*. Qualche esempio:

```
puts 4
puts 1+3
```

```
>4
>4
>
```

La funzione `puts`

```
puts arg1, arg2, arg3, …, argn
```

Produce in STDOUT ognuno degli argomenti passati dopo averli convertiti in stringa di caratteri, stampando dopo ogni argomento un carattere di *newline*. Qualche esempio:

```
puts 4
puts 1+3
```

```
>4
>4
>
```

Attenzione! Se si scrivono le due istruzioni su una sola riga

```
puts 4 puts 1+3
```

il codice non funziona…

La funzione `puts`

```
puts arg1, arg2, arg3, …, argn
```

Produce in STDOUT ognuno degli argomenti passati dopo averli convertiti in stringa di caratteri, stampando dopo ogni argomento un carattere di *newline*. Qualche esempio:

```
puts 4
puts 1+3
```

```
>4
>4
>
```

Se si vogliono due istruzioni su una sola riga occorre aggiungere un ‘;’

```
puts 4; puts 1+3
```

La funzione `puts`

```
puts arg1, arg2, arg3, …, argn
```

Produce in STDOUT ognuno degli argomenti passati dopo averli convertiti in stringa di caratteri, stampando dopo ogni argomento un carattere di *newline*. Qualche esempio:

```
puts “Risultato =“, 1+3
```

```
>Risultato =
>4
>
```

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato e restituito è un oggetto
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato e restituito è un oggetto
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi
- Tutto è considerato vero tranne il valore booleano falso (`false`) e il valore nullo (`nil`)

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato e restituito è un oggetto
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi
- Tutto è considerato vero tranne il valore booleano falso (`false`) e il valore nullo (`nil`)
- I commenti iniziano con # e terminano alla fine della riga

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato e restituito è un oggetto
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi
- Tutto è considerato vero tranne il valore booleano falso (`false`) e il valore nullo (`nil`)
- I commenti iniziano con # e terminano alla fine della riga
- L’indentazione (seguendo il Ruby-style) è di 2 caratteri

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato e restituito è un oggetto
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi
- Tutto è considerato vero tranne il valore booleano falso (`false`) e il valore nullo (`nil`)
- I commenti iniziano con # e terminano alla fine della riga
- L’indentazione (seguendo il Ruby-style) è di 2 caratteri
- Il valore delle variabili è sempre un riferimento ad un oggetto (anche quelle che contengono un valore numerico o una stringa, in quanto anche numeri e stringhe in Ruby sono oggetti).

Le “cose” da sapere su Ruby

- Le variabili non vengono dichiarate: incominciano ad esistere nel momento in cui si assegna loro un valore

Le “cose” da sapere su Ruby

- Le variabili non vengono dichiarate: incominciano ad esistere nel momento in cui si assegna loro un valore
- Le variabili non sono tipizzate: il loro tipo viene determinato automaticamente nel momento dell’assegnamento

```
a = 1245
puts a
```

```
>1245
>
```

Le “cose” da sapere su Ruby

- Le variabili non vengono dichiarate: incominciano ad esistere nel momento in cui si assegna loro un valore
- Le variabili non sono tipizzate: il loro tipo viene determinato automaticamente nel momento dell’assegnamento

```
a = 1245
puts a
```

```
>1245
>
```

- Il ‘;’ che termina un’istruzione è opzionale se questa è l’ultima istruzione sulla riga

Le “cose” da sapere su Ruby

- Le parentesi tonde attorno ai parametri attuali nell’invocazione di un metodo/funzione sono opzionali (tranne quando servono per disambiguare invocazioni annidate)

Le “cose” da sapere su Ruby

- Le parentesi tonde attorno ai parametri attuali nell’invocazione di un metodo/funzione sono opzionali (tranne quando servono per disambiguare invocazioni annidate)

```
puts “Hello”
```

equivale a

```
puts(“Hello”)
```

I tipi standard

- Valori booleani
  - `true` e `false`

```
b = true
puts b
```

```
>true
>
```

I tipi standard

- Valori booleani
  - `true` e `false`

```
b = true
puts b
```

```
>true
>
```

- Numeri interi

```
intero = 1301
puts intero
```

```
>1301
>
```

I tipi standard

- Numeri decimali

```
decimale = 13.01
puts decimale
```

```
>13.01
>
```

I tipi standard

- Numeri decimali

```
decimale = 13.01
puts decimale
```

```
>13.01
>
```

- Numeri complessi
  - composti da una parte reale e una parte immaginaria
  - esempio: *2+3i*

I tipi standard

- Numeri decimali

```
decimale = 13.01
puts decimale
```

```
>13.01
>
```

- Numeri complessi
  - composti da una parte reale e una parte immaginaria
  - esempio: *2+3i*
- Numeri razionali
  - frazioni = rapporto tra due numeri interi
  - esempio: *3/4*

I tipi standard

- Stringhe
  - sequenze di caratteri

```
stringa = “Hello world!”
puts stringa
```

```
>Hello world!
>
```

Valori booleani

I valori booleani sono true e false (vero o falso) e vengono creati attraverso i letterali `true` e `false`.

```
true_value = true
false_value = false
```

`true_value` è un oggetto della classe `TrueClass`, mentre `false_value` è della classe `FalseClass`.

Numeri interi

I numeri interi sono di qualsiasi lunghezza (dipende dalla memoria disponibile sul sistema); fino ad un certo valore (numeri “piccoli”) sono oggetti della classe `Fixnum`, oltre (numeri “grandi”) sono oggetti della classe `Bignum`

```
small_int = 245667589600
big_int = 24566758960089658688888888888
new_int = small_int*big_int
small_int = small_int*big_int
```

- `small_int` è un oggetto della classe `Fixnum`, mentre `big_int` è della classe `Bignum`.
- `new_int` (prodotto di un `Fixnum` e di un `Bignum`) è di classe `Bignum`.
- `small_int` dopo l’ultima istruzione diventa un `Bignum`.

Numeri interi

I numeri interi vengono creati attraverso un letterale (sequenza di simboli) composto da:

- cifre da 0 a 9
- simbolo underscore '_' (che viene ignorato)
- simbolo '-' all'inizio che effettua la negazione aritmetica

```
num1 = 245667
num2 = 245_667
num3 = -245667
puts num1
puts num2
puts num3
```

```
>245667
>245667
>-245667
>
```

Di default tali letterali vengono interpretati come interi in base 10

Numeri interi

I numeri interi vengono creati attraverso un letterale (sequenza di simboli) composto da:

- cifre da 0 a 9
- simbolo underscore '_' (che viene ignorato)
- simbolo '-' all'inizio che effettua la negazione
- simboli opzionali all'inizio del letterale che possono essere:
  - 0 se il letterale seguente esprime un intero in base 8
  - 0x se il letterale seguente esprime un intero in base 16
  - 0b se il letterale seguente esprime un intero in base 2

Numeri interi

```
num = 84919      #intero in base 10
puts num
```

```
>84919
>
```

```
num = 0245667    #intero in base 8
puts num
```

```
>84919
>
```

Numeri interi

```
num = 0b100111      #intero in base 2
puts num
```

```
>39
>
```

```
num = 0x456afb      #intero in base 16
puts num
```

```
>4549371
>
```

Numeri decimali

I numeri decimali sono oggetti della classe `Float`

```
decimal1 = 1356.564
```

e vengono creati attraverso un letterale (sequenza di simboli) composto da:

- cifre da 0 a 9
- simbolo underscore '_' (che viene ignorato)
- simbolo '.' come separatore
- simbolo '-' all'inizio che effettua la negazione aritmetica
- simbolo 'e' per l'esponente

```
decimal2 = 1.34e3       #1340
```

Numeri decimali

Attenzione! Il separatore '.' deve essere preceduto e seguito da una cifra

```
decimal3 = 1.e3  #Intendendo il numero 1000, non
                 #funziona perché si cercherebbe
                 #di accedere al metodo e dell'oggetto
                 #1 che viene interpretato come Fixnum
```

Attenzione!

```
num = 10e2    #Viene interpretato come decimale e NON
              #intero; è quindi un oggetto Float
```

Classe `Numeric`

Tutti i numeri (anche i razionali e i complessi) sono in generale istanze della classe `Numeric`, che mette a disposizione metodi quali:

- `abs`: restituisce il valore assoluto del numero
- `ceil`: restituisce il più piccolo intero maggiore o uguale al numero
- `floor`: restituisce il più grande intero minore o uguale al numero
- `integer?`: restituisce `true` se il numero è intero, altrimenti `false`
- `zero?`: restituisce true se il numero è uguale a 0, altrimenti false
- `to_int`: converte il numero in un intero

Stringhe

Una stringa è una sequenza di simboli

```
stringa1 = “Hello world!”
stringa2 = ‘Ciao mondo!’
```

e viene creata attraverso un letterale (sequenza di simboli) composto da doppi apici “ o singoli apici ‘ come delimitatori (non fanno parte del valore della stringa).
Di default Ruby è basato sul codice ASCII; se si vogliono includere anche i simboli di Unicode, si deve inserire all’inizio del codice la riga: `# encoding: utf-8`

```
# encoding: utf-8

puts “Hello world!”
```

Stringhe

- Tra singoli apici
  - avviene la sostituzione di `\\` con `\` e di `\'` con `'`
  - si può inserire il simbolo "

```
puts '"Hello" \'world\''
```

- Tra doppi apici
  - avviene la sostituzione di `\n` con il *newline*

```
puts "Hello\nworld", 'Hello\nworld'
```

Stringhe

- Tra singoli apici
  - avviene la sostituzione di `\\` con `\` e di `\'` con `'`
  - si può inserire il simbolo "

```
puts '"Hello" \'world\''
```

```
>"Hello" 'world'
>
```

- Tra doppi apici
  - avviene la sostituzione di `\n` con il *newline*

```
puts "Hello\nworld", 'Hello\nworld'
```

Stringhe

- Tra singoli apici
  - avviene la sostituzione di `\\` con `\` e di `\'` con `'`
  - si può inserire il simbolo “

```
puts '"Hello" \'world\''
```

```
>"Hello" 'world'
>
```

- Tra doppi apici
  - avviene la sostituzione di `\n` con il *newline*

```
puts "Hello\nworld", 'Hello\nworld'
```

```
>Hello
>world
>Hello\nworld
>
```

Stringhe

- Tra doppi apici si può usare il costrutto `#{expr}` per sostituire il risultato della valutazione di una qualsiasi espressione *expr*

```
a = 1789
b = 675
puts “Il risultato di a+b e’ #{a+b}“
puts “Il risultato di #{a}+#{b} e’ #{a+b}“
puts ‘#{a} non viene sostituito tra singoli apici \’’
```

Stringhe

- Tra doppi apici si può usare il costrutto `#{expr}` per sostituire il risultato della valutazione di una qualsiasi espressione *expr*

```
a = 1789
b = 675
puts “Il risultato di a+b e’ #{a+b}“
puts “Il risultato di #{a}+#{b} e’ #{a+b}“
puts ‘#{a} non viene sostituito tra singoli apici \’’
```

```
>Il risultato di a+b e’ 2464
>Il risultato di 1789+675 e’ 2464
>#{a} non viene sostituito tra singoli apici ‘
>
```

Stringhe

Qualcosa in più sulle stringhe:

- I caratteri in una stringa vengono indicizzati a partire da 0
- Si può accedere al singolo carattere in una stringa attraverso la sintassi *`string_name`*`[`*`index`*`]`, dove *`index`* è l'indice del carattere all'interno della stringa
- La stringa vuota è la stringa "" oppure ''
- Un singolo carattere è trattato come una stringa di lunghezza 1
- Le stringhe sono oggetti della classe `String`

```
str = "Hello world!"
puts str
str[0] = "h"
puts str
```

```
>Hello world!
>hello world!
>
```

Stringhe

Qualcosa in più sulle stringhe:

- La somma di due stringhe produce la loro concatenazione
- Il prodotto di una stringa con un intero n (e non viceversa) produce la ripetizione della stringa per n volte

```
str1 = “Hello”
str2 = “ world!”
puts str1+str2
str3 = str1*3
puts str3
```

```
>Hello world!
>HelloHelloHello
>
```

Metodi utili di `String`...

```
string_name.downcase!
```

Il metodo `downcase!` mette la stringa in minuscolo

```
string_name.upcase!
```

Il metodo `upcase!` mette la stringa in maiuscolo

```
stringa = “HELLO”
stringa.downcase!
puts stringa
```

```
>hello
>
```

Stringhe

Un altro modo di costruire una stringa è il costrutto *HERE DOCUMENT* che permette di costruire facilmente stringhe su più righe.

```
string_name = <<END_TAG
    La stringa è formata dalle righe fino a quella che
    inizia con lo stesso tag specificato dopo <<
END_TAG
```

END_TAG deve stare a inizio riga a meno che dopo i simboli << non venga specificato un simbolo -.

```
string_name = <<-END_TAG
    La stringa è formata dalle righe fino a quella che
    inizia con lo stesso tag specificato dopo <<
    END_TAG
```

Stringhe

Un altro modo di costruire una stringa è il costrutto *HERE DOCUMENT* che permette di costruire facilmente stringhe su più righe.

```
stringa = <<FINE_STRINGA
   La stringa è formata dalle righe fino a quella
   che inizia con lo stesso tag specificato dopo <<
FINE_STRINGA
puts stringa
```

```
>   La stringa è formata dalle righe fino a quella
>   che inizia con lo stesso tag specificato dopo <<
>
```