Introduzione al Python

Andrea Passerini
passerini@disi.unitn.it

Informatica

Introduzione al Python

Caratteristiche

procedurale si specifica la procedura da eseguire sui dati

strutturato concetto di visibililtà delle variabili

orientato agli oggetti permette di programmare con le classi

interpretato il programma viene eseguito da un interprete e non direttamente dalla CPU (in realtà viene prima compilato in bytecode e poi eseguito da una Python Virtual Machine)

Perché Python

- Essendo un linguaggio interpretato, Python è facile da imparare ed usare, grazie all'interazione dinamica con l'interprete
- Rispetto ad altri linguaggi interpretati (e.g. Perl), fornisce maggiori funzionalità rendendolo più simile ad un linguaggio compilato, ed è più intuitivo
- **E' un linguaggio estremamente diffuso in bioinformatica, grazie ad un insieme di librerie specifiche (*BioPython*)**

Introduzione al Python: installazione dell'interprete

GNU/Linux

- l'interprete Python è generalmente preinstallato nelle distribuzioni recenti
- Aprite un interprete di comandi, lanciate l'interprete Python con:

```
python
```

- Nel caso non sia installato, può essere installato da sorgenti o tramite un gestore di pacchetti (a seconda della distribuzione GNU/Linux)

Introduzione al Python: installazione dell'interprete

Mac OS X

- Un interprete Python è preinstallato a partire da Mac OS X 10.2
- Aprite un interprete di comandi, lanciate l'interprete Python con:

```
python
```

- Nel caso non sia installato, scaricate il programma di installazione da `http://www.python.org/download/`

Introduzione al Python: installazione dell'interprete

Windows

- Windows non fornisce alcun interprete Python preinstallato.
- Scaricate il programma di installazione da `http://www.python.org/download/`
- Avviate l'interprete Python selezionandolo dal menu dei programmi

Uso interattivo dell'interprete

- L'interprete di comandi Python permette di eseguire interattivamente comandi Python e visualizzarne il risultato
- Una volta avviato, l'interprete presenta un cursore in cui inserire comandi
- Scrivendo un comando Python e premendo `invio`, l'interprete esegue il comando e ne visualizza il risultato
- In pratica l'interprete stampa su schermo una stringa che rappresenta l'output del comando inserito

Uso interattivo dell'interprete

Esempio (GNU/linux)

```
> python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| ...
[GCC 4.2.1 Compatible Apple LLVM 6.0 ...
Type "help", "copyright", "credits" or ...
>>> 2 * 4
8
>>> print(2 * 4)
8
>>>
```

Uso interattivo dell'interprete

vantaggi

- L'uso interattivo dell'interprete permette di verificare velocemente l'output di un comando
- E' un modo rapido ed efficace di testare codice Python
- E' un modo rapido di accedere alla documentazione di comandi e funzioni (vedremo)
- E' un modo rapido di verificare il contenuto di librerie di codice Python (vedremo)
- L'interprete ricorda (history) i comandi inseriti durante una sessione (fino alla chiusura dell'interprete), e possono essere recuperati per rieseguirli (e.g. $\uparrow$ in GNU/Linux)

Uso interattivo dell'interprete

svantaggi

- L'interprete non salva i comandi inseriti su file.
- L'uso della history è limitato alla sessione stessa
- L'interprete non è adatto a scrivere programmi grandi o che debbano essere riutilizzati

Scrittura di programmi su file

- La programmazione vera e propria viene fatta creando dei file di comandi (*programmi* o *script*)
- Un file di comandi Python può essere creato con qualsiasi editor di testi
- Per poter essere importati come moduli (vedremo), i file Python devono avere l'estensione `.py` (attenzione agli editor che aggiungono estensioni automatiche ai file salvati)
- Un file di comandi può essere eseguito in modalità *batch* tramite l'interprete Python
- L'inteprete esegue i comandi contenuti nel file uno dopo l'altro e termina.
- A differenza dell'uso interattivo, in questo caso l'interprete non stampa automaticamente l'output dei comandi (usare esplicitamente `print` a questo scopo)

Scrittura di programmi su file

Esempio (GNU/linux)

- Creazione di un file `prova.py` (il testo dopo '#' sono commenti ignorati dall'interprete):

```
2*8             # esegue 2*8 (no output)
print(2 * 6)    # stampa risultato di 2 * 6
a = 2 * 5       # assegna ad a risultato di 2 * 5
print(a)        # stampa il contenuto di a
```

- Esecuzione del file tramite l'interprete:

```
> python prova.py
12
10
```

Oggetti

Ogni cosa è un oggetto

- Python manipola oggetti
- Ad ogni oggetto è associato un *tipo*.
- Python fornisce una serie di tipi di oggetti predefiniti. I principali sono:

numeri	`6.345`
stringhe	`"The meaning of life"`
liste	`["a", "b", 23]`
dizionari	`{ "UNO" : 1, "DUE" : 2}`
tuple	`("basso","medio","alto")`
file	`myfile = fopen("file.txt", 'r")`
None	indica l'oggetto "nullo"

- Gli oggetti possono essere manipolati tramite operatori (`5.3 + 4.2`) e funzioni (`print("print me")`)

Creazione di oggetti

- Un oggetto si crea “assegnando” un valore ad una *variabile* che rappresenterà un *riferimento* all’oggetto stesso:

```
>>> a = 12 * 3
>>> b = "sono una stringa"
```

- Il tipo dell’oggetto viene stabilito al momento della creazione, e dipende dal valore che gli viene assegnato:

```
>>> a = 12 * 3                    # number
>>> b = "sono una stringa"        # string
```

Operazioni su oggetti

- Le operazioni che possono essere fatte (ed il loro risultato) dipendono dal tipo degli oggetti coinvolti:

```
>>> a = 4
>>> b = a * 3                    # number
>>> print(b)
12
>>> c = "sono"
>>> d = " una stringa"
>>> e = c + d                    # string
>>> print(e)
sono una stringa
```

Variabili

Identificatori di oggetti

- Una variabile rappresenta un riferimento ad un oggetto in un programma Python
- Una variabile è una sequenza arbitraria di:
  - lettere maiuscole ([A..Z])
  - lettere minuscole ([a..z])
  - cifre ([0..9])
  - underscore (_)

  in cui il primo carattere non sia una cifra

Esempi

- Corrette:

```
a   b1 _c4   A45b   var_
```

- Scorrette:

```
12a      # comincia per cifra
var$     # carattere non ammesso
```

Variabili

Scelta del nome

- E' utile dare alle variabili un nome che ricordi la loro funzione (`year = 2008`)
- Nomi troppo lunghi possono essere scomodi se la variabile deve essere scritta spesso nel programma

Keywords

Python ha una serie di keyword *riservate* che non possono essere usate come identificatori (vedremo):

```
and       del       from    not     while
as        elif      global  or      with
assert    else      if      pass    yield
break     except    import  print
class     exec      in      raise
continue  finally   is      return
def       for       lambda  try
```

Variabili

Tipizzazione dinamica

- Python utilizza una forma di tipizzazione *dinamica*:
  - Solo gli oggetti hanno tipi
  - il tipo di una variabile è il tipo dell'oggetto a cui si riferisce
- se ad una variabile si assegna un nuovo oggetto, il suo tipo diventa quello del nuovo oggetto a cui si riferisce (mentre il vecchio oggetto rimane invariato)

```
>>> a = 4                  # number
>>> a = "stringa"          # string
>>> b = a
>>> print(b)
stringa
>>> b = 3
>>> print(b)
3
>>> print(a)
stringa
```

Variabili

Tipizzazione forte

- In ogni istante, una variabile ha il tipo dell'oggetto a cui si riferisce
- Non è possibile eseguire operazioni sulla variabile che non siano compatibili con il suo tipo (non c'è conversione implicita di tipo)

```
>>> a = 4                  # number
>>> b = "stringa"          # string
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
'int' and 'str'
```

Commenti

Descrizione

- Un commento comincia con il carattere '#'
- Tutto ciò che segue fino al termine della riga viene ignorato dall'interprete:

```
>>> a = "aa"
>>> a = a * 2          # replica due volte a
>>> print(a)           # stampa a
aaaa
```

- I commenti servono a favorire la comprensione del codice dopo averlo scritto, o da parte di altri.

Statements

Descrizione

- Uno *statement semplice* è un comando contenuto in una unica riga
- Esistono vari tipi di statement. Finora abbiamo visto:

  assegnazione `x = a + b`
  espressione `a + b` (principalmente per uso interattivo)
  funzione `print(a + b)`

- Esistono anche *statement composti* che operano su più righe (tipicamente per uso non-interattivo, vedremo):

```
if a > b:
    tmp = b
    b = a
    a = tmp
```

Funzioni

Descrizione

- Una funzione è una sequenza di comandi raggruppati in modo da poter essere chiamati più volte (un tipo particolare di statement composto)
- Una funzione ha un nome con il quale viene chiamata (stesse limitazioni del nome di variabile)
- Una funzione piò avere uno o più oggetti in ingresso (specificati tra parentesi dopo il nome)
- Una funzione può produrre un oggetto in uscita

Esempi

```
>>> len([1,2,3])    # calcola la lunghezza di una lista
3
>>> f = open("file.txt", "r") # apre un file in lettura
```

Funzioni

Funzioni disponibili

- Python fornisce una serie di funzioni di utilità *predefinite* (e.g. `len, open`)
- Gli oggetti (a parte i numeri) hanno funzioni proprie per manipolarli (vedremo)
- E' possibile scrivere le proprie funzioni (vedremo)

Moduli

Collezioni di codice

- Programmi complessi vengono suddivisi in componenti chiamati *moduli*
- Un modulo corrisponde ad un file di codice Python (estensione ".py")
- I moduli forniscono una vasta gamma di funzioni di utilità (ed altro) che possono essere impiegate nei propri programmi
- Per poter utilizzare il contenuto di un modulo, deve essere *importato* (comando `import`):

```
>>> import re  # modulo per espressioni regolari
>>> re.findall("[CHDE]","AAGCFDDHGDEC")
['C', 'D', 'D', 'H', 'D', 'E', 'C']
```