Introduzione a Python

31 gennaio 2018

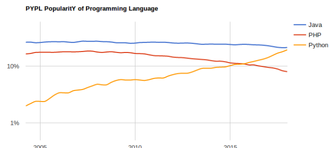
Dott. Ing. Roberto Pasolini, PhD
roberto.pasolini@unibo.it

Python

- Linguaggio interpretato cross-platform
  - disponibile per i principali SO (Linux, Mac, Windows, ...)
  - un'implementazione di riferimento (*CPython*), più altre alternative (*PyPy*, *Jython*, ...)
- Multi-paradigma
  - imperativo, object-oriented, funzionale
- Enfasi sulla facilità di lettura e scrittura del codice

Perché Python?

- Linguaggio general-purpose, usato per molteplici scopi
  - scripting, sviluppo Web, data science, ...
- Facile da imparare
- Usato per cicli di sviluppo rapidi
- Popolarità in rapido aumento negli ultimi anni →
- Libreria standard molto completa
- Ampia disponibilità di librerie esterne



Trend ricerche su Google di tutorial sui linguaggi (da http://pypl.github.io/*)*

Python 2 e Python 3

- Sono diffuse due diverse versioni *major* di Python
- Su **Python 2** si basa molto software tutt'ora in uso
  - utilizzato di default ad es. in Ubuntu e derivati
  - l'ultima versione *minor* prevista è la 2.7, rilasciata nel 2010
  - il termine dello sviluppo è previsto nel 2020
- **Python 3** introduce alcune incompatibilità con Python 2
  - prima release nel 2008, ultima versione minor 3.6 del 2016
- Molte librerie di uso comune sono state (ri)scritte per funzionare con entrambi
- **Nel corso useremo Python 3**

Le differenze chiave con Python 2 saranno indicate in riquadri come questo

Sintassi di base

- Un'istruzione Python è normalmente contenuta in una riga
  - non è necessario terminare la riga con ";" o altro

```
print("Hello, world")
```

- I commenti sono introdotti da "#" e finiscono a fine riga

```
# Questo e un commento
print("Hello, world")    # altro commento
```

- Si possono usare righe vuote per separare parti di codice
- Usare ";" per separare più istruzioni in una riga

```
print("Hello"); print("world")
```

Sintassi di base: istruzioni su più righe

Un'istruzione può continuare in una riga successiva

- *esplicitamente* se la riga termina con "\"

```
num = 2 \
    + 2
```

- *implicitamente* se ci sono parentesi aperte (più comune)

```
num = ( 2   # parentesi aperta -> continua sotto
      + 2 )
print("Hello, "
    + "world!")
```

La guida di stile ufficiale (PEP 8) prevede:

- *righe non più lunghe di 79 caratteri*
- *operatori binari affiancati all'operando di destra (come sopra)*

Sintassi di base: blocchi di codice

- In altri linguaggi i blocchi di codice (usati in costrutti `if`, `for`, ...) sono delimitati da caratteri specifici (spesso `{ }`)
  - l'indentazione è usata per migliorare la leggibilità
- Python usa l'indentazione per riconoscere i blocchi
  - ogni riga prima di un blocco indentato termina in ":"
  - ogni riga di un blocco è indentata con ugual numero di spazi
  - la prima riga dopo il blocco è allineata a quelle precedenti

```
// esempio in Java
x = getValue();
if (x < 0) {
    x = 0;
}
System.out.println(x);
```

```
# esempio in Python
x = get_value()
if x < 0:
    x = 0
println(x)
```

Interprete interattivo

- Python può essere eseguito in modalità interattiva
  - l'utente digita espressioni una ad una
  - l'interprete le valuta e ne stampa il risultato
  - noto anche come REPL (*Read-Eval-Print Loop*)
- Per avviare l'interprete in questa modalità
  - Linux/Mac: comando `python3` (senza specificare un file)
  - Windows: Start → Python 3.x → IDLE
- Per uscire dall'interprete, usare `exit()` o (su Linux/Mac) premere Ctrl+D
- Nel codice riportato nelle slide successive, “>>>” rappresenta il prompt dell'interprete

Oggetti e tipi in Python

- In Python ogni cosa è *oggetto*: numeri, stringhe, funzioni, . . .
    - al contrario di Java, dove ad es. i numeri sono trattati diversamente
- Ogni oggetto ha un tipo che ne determina le operazioni supportate
- I tipi degli oggetti sono noti solo durante l'esecuzione
    - al contrario di Java, non si hanno informazioni sulla compatibilità tra tipi durante la scrittura o la compilazione
- Ogni oggetto ha *attributi*, a cui si può accedere con la sintassi "*oggetto.attributo*" come in Java

Tipi di base: numeri

- Python supporta numeri interi (tipo `int`) e frazionari (tipo `float`) di precisione arbitraria
- Sono supportate le ordinarie operazioni aritmetiche: `+ - * /`
  - Operazioni tra `int` e `float` restituiscono sempre `float`
  - La divisione con `/` da sempre `float`
  - Si possono usare le parentesi `( )`
- Altri opearatori:
  - `%`: modulo (intero positivo)
  - `//`: divisione intera
  - `**`: elevamento a potenza

```
>>> 2 + 2
4
>>> (4-1.5) * 4
10.0
>>> 6 / 3
2.0
>>> 13 // 5
2
>>> 13 % 5
3
>>> 2 ** 8
256
```

Tipi di base: booleani

- Python definisce le costanti `True` e `False` di tipo `bool`
  - va usata l'iniziale maiuscola!
- Si possono applicare gli operatori logici `and`, `or` e `not`
- Sono il risultato di operazioni di comparazione `==` `!=` `>` `<` `>=` `<=`
- Si possono concatenare più comparazioni (es. per verificare che un numero sia compreso tra altri due)

```
>>> not True
False
>>> True and False
False
>>> 2 == 2
True
>>> 3 != 3
False
>>> 4 >= 4
True
>>> 0 < 42 < 100
True
```

Variabili

- Le variabili in Python non sono dichiarate esplicitamente: per "crearne" una basta assegnargli un valore
- Per l'assegnamento si usa la tipica sintassi "*nome = valore*"

```
>>> foo = 21 * 2
>>> foo
42
>>> 5 * foo
210
```

- Le variabili **non** hanno tipo: è possibile sostituire un valore con un altro di tipo diverso

```
>>> foo = 12.345
>>> foo = False
```

Variabili: assegnamento esteso

- Anche Python fornisce operatori per assegnare ad una variabile il risultato di un'operazione sul suo valore

```
>>> foo = 40
>>> foo += 2    # equivale a foo = foo + 2
>>> foo
42
```

- Questo funziona in generale per qualsiasi operatore e qualsiasi tipo di dato lo supporti
- In Python **non** esistono però gli operatori ++ e -- per modificare e leggere una variabile

Funzioni

- Python definisce diverse funzioni standard, richiamabili con la tipica sintassi *nome*(*parametri*)
- La funzione `print` stampa un oggetto in output
  - si possono passare più parametri per stampare più oggetti, separati da spazi

```
>>> print("Hello, world!")
Hello, world!
>>> print("Hello", "world", 42)
Hello world 42
```

In Python 2, `print` è un'istruzione del linguaggio utilizzabile senza parentesi

Stringhe

- Una stringa (tipo `str`) è delimitata tra apici singoli o doppi
  - ad es. `'hello'` e `"hello"` sono identiche
  - se una stringa contiene uno dei due caratteri, conviene usare l'altro per delimitare, es. `"don't"`
  - nel corso useremo come convenzione i doppi apici
- Si possono usare le tipiche sequenze di *escape* con “\”
  - ad es. "`\n`" = a capo, "`\t`" = tabulazione, “\\” = “\”
  - per **non** interpretare gli escape, usare “r” davanti alla stringa:

```
>>> print("Hello\nworld")
Hello
world
>>> "C:\\Users\\Bob" == r"C:\Users\Bob"
True
```

Stringhe multiriga

- Per scrivere una stringa su più righe, utilizzare tre apici (singoli o doppi)

```
>>> print("""Hello,
... world!""")
Hello,
world!
```

- Usare "\" a fine riga per **non** andare a capo nel risultato

```
>>> print("""Hello, \
... world!""")
Hello, world!
```

- Anche le stringhe con tre apici possono contenere sequenze di escape o possono essere marcate con "r" per ignorarle

Liste

- Un oggetto di tipo `list` rappresenta una sequenza di oggetti
  - gli oggetti possono essere di tipi eterogenei
- Si crea una lista di oggetti usando le parentesi quadre

```
>>> names = ["Alice", "Bob", "Carol"]
>>> stuff = [42, -12.345, True, "Hello"]
>>> nothing = []
```

- Le operazioni viste per le stringhe (`+`, `len`, `[]`) sono valide anche per le liste

```
>>> len(names)
3
>>> names[1]
'Bob'
```

Liste: modifica

- Al contrario degli altri oggetti visti finora, le liste sono mutabili: una volta create il loro valore può cambiare
- Si può sostituire un oggetto assegnandolo ad una posizione specifica
- Si può aggiungere un oggetto in coda a una lista chiamandone il metodo `append`
- Si può rimuovere un oggetto dalla lista con l'istruzione `del`

```
>>> nums = [4, 2]

>>> nums[0] = 1
>>> nums
[1, 2]

>>> nums.append(3)
>>> nums
[1, 2, 3]

>>> del nums[1]
>>> nums
[1, 3]
```

Tuple

- Gli oggetti `tuple` sono sequenze di oggetti simili alle `list`, ma immutabili
  - non si possono sostituire, aggiungere o rimuovere elementi
  - si possono però modificare oggetti mutabili al loro interno!
- Si creano come le liste, ma usando parentesi tonde al posto delle quadre
  - le parentesi sono opzionali, se non si crea ambiguità
  - una tupla con un solo elemento si crea aggiungendo una virgola dopo di esso
- Sulle tuple si possono usare le stesse operazioni delle liste (+, `len`, ...)

```
>>> x = (42, True)
>>> y = 42, False
>>> single = (8, )
>>> single2 = 8,
>>> nothing = ()

>>> foo[1]
True

>>> len(single)
1
```

Insiemi (set)

- Gli oggetti set sono insiemi mutabili di elementi immutabili
  - al contrario delle liste, un elemento può apparire una sola volta e non ha una posizione specifica nell'insieme
  - l'aggiunta di un oggetto mutabile ad un insieme da errore
- Per creare un insieme si usano le parentesi graffe
  - per un insieme vuoto bisogna scrivere "set()"!

```
>>> fruits = {"apple", "banana", "orange"}
>>> nothing = set()
```

- I metodi add e remove inseriscono e tolgono elementi

```
>>> fruits.add("lemon")
>>> fruits.remove("orange")
>>> fruits
{'banana', 'lemon', 'apple'}
```

Dizionari (dict)

- Gli oggetti dict sono delle associazioni tra chiavi e valori
  - simili agli oggetti Map in Java
- Le chiavi sono oggetti immutabili di qualsiasi tipo
- Un dizionario si crea specificando tra parentesi graffe una sequenza di coppie "*chiave* : *valore*"

```
>>> prices = {"water": 1.5, "soda": 2, "beer": 2.5}
>>> nothing = {}
```

- Si può accedere al valore associato ad una chiave data
  - se alla chiave non è associato nulla, si ha un NameError

```
>>> prices["beer"]
2.5
```

Dizionari: modifica

- Gli oggetti `dict` sono mutabili
- Si possono assegnare valori sia a chiavi già esistenti (sostituendo il precedente) che nuove

```
>>> prices = {"water": 1, "soda": 2}
>>> prices["water"] = 1.5
>>> prices["beer"] = 2.5
>>> prices
{'water': 1.5, 'soda': 2, 'beer': 2.5}
```

- Si può rimuovere un'associazione con `del`

```
>>> del prices["water"]
>>> prices
{'soda': 2, 'beer': 2.5}
```

Operazioni sulle collezioni

- Liste, tuple, insiemi e dizionari rappresentano in generale *collezioni* di elementi
- Anche le stringhe sono considerate collezioni di caratteri
  - Python non definisce un tipo di dato "carattere", usa invece stringhe di lunghezza unitaria
- Esistono diverse operazioni comuni eseguibili su tali oggetti
- La funzione `len` restituisce il numero di elementi

```
>>> len("Hello")
5
>>> len([42, -12.345, "Hello", False])
4
>>> len({5: "foo", 8: "bar"})
2
```

Operazioni sulle collezioni: `in`

- L'operatore `in` verifica la presenza di un elemento
  - esiste anche la versione negata `not in`
- Nei dizionari sono controllate le chiavi, **non** i valori
- Nelle stringhe si possono cercare sequenze di caratteri di lunghezza arbitraria

```
>>> "B" in ["A", "B", "C"]
True
>>> "foo" in {"foo": "bar"}
True
>>> "bar" not in {"foo": "bar"}
True
>>> "lo" in "hello"
True
```

Operazioni sulle sequenze

- Per strutture dati sequenziali è possibile accedere agli elementi in base alla loro posizione
  - stringhe, liste e tuple, ma non insiemi e dizionari
- Per estrarre un elemento, si indica l'indice tra quadre
  - il primo elemento ha indice 0

```
>>> foo = "Hello"
>>> foo[1]   # secondo carattere della stringa
'e'
```

- Usando un indice negativo $-n$, si accede al n-ultimo elemento

```
>>> foo[-1]   # ultimo carattere della stringa
'o'
```

Operazioni sulle sequenze: intervalli

- Per estrarre elementi in un intervallo, si usa al posto dell'indice la notazione *inizio* : *fine*
    - l'indice iniziale è incluso, quello finale è escluso
    - se non indicati, si considerano inizio e fine sequenza
    - si può aggiungere ":passo" per "saltare" alcuni elementi

```
>>> foo = "ABCDEFGHI"
>>> foo[2:4]    # dal terzo al quarto carattere
'CD'
>>> foo[:3]    # primi tre caratteri
'ABC'
>>> foo[-3:]    # ultimi tre caratteri
'GHI'
>>> foo[4::2]    # ogni due caratteri dal quinto
'EGI'
```

Operazioni sulle sequenze: modifica

- Nelle sequenze modificabili (`list`) è possibile assegnare un valore ad una posizione specifica
  - È possibile sostituire anche un intero intervallo

```
>>> foo = ["A", "X", "Y", "Z", "F"]
>>> foo[1] = "B"    # sostituisci secondo elemento
>>> foo
['A', 'B', 'Y', 'Z', 'F']
>>> foo[2:4] = ["C", "D", "E"]
>>> foo
['A', 'B', 'C', 'D', 'E', 'F']
```

Controllo di flusso: `if-elif-else` e `while`

- Il costrutto `if` permette di eseguire un blocco di istruzioni condizionalmente
- Ad esso si possono aggiungere una o più clausole `elif` (*else if*) e/o una clausola `else`
- Il costrutto `while` permette di ripetere un blocco di istruzioni finché una condizione è verificata

```
if vote == 30:
  msg = "Perfect!"
elif vote >= 18:
  msg = "Passed"
else:
  msg = "Retry"
println(msg)

n = 0
while n < 10:
  print(n)
  n += 1
```

Controllo di flusso: `for`

- Tutte le collezioni (sequenziali e non) sono oggetti *iterabili*: è possibile scorrere in sequenza i loro contenuti
  - in seguito vedremo altri tipi di oggetti iterabili
- Il costrutto `for` esegue un blocco di codice una volta per ogni elemento estratto da un iterabile dato

```
foo = [1, 5, 21]
# stampa i doppi di ogni elemento nella lista
for x in foo:
    print(x*2)
```

Controllo di flusso: `break`, `continue`, `else`

- In un ciclo `while` o `for`, un'istruzione `break` provoca l'uscita immediata dal ciclo
- L'istruzione `continue` causa invece l'interruzione dell'iterazione corrente e l'inizio della successiva (se c'è)
- In Python si può aggiungere al ciclo un blocco `else`, eseguito a fine ciclo se e solo se **non** è stato interrotto da `break`

```
for x in numbers:    # con numbers lista di numeri
    if x < 0:
        println("Found negative number")
        break    # salta il blocco else
else:
    # break non eseguito
    println("No negative number found")
```

Definizione di funzioni

- Per definire una funzione si usa il blocco `def`
- Una funzione può accettare dei parametri
- Una funzione può restituire un valore usando `return`

```
def factorial(x):
    if x > 1:
        return x * factorial(x-1)
    else:
        return 1
```

- Se non si usa `return`, la funzione restituisce implicitamente la costante `None` (simile a `null` in Java)