

Laboratorio di Python

Definizioni di Variabile e Funzione in Python

08 Marzo 2019

Variabili modificabili

- Nomi/etichette per valori
- **Valori modificabili:** possono cambiare nel tempo (\neq dalla Matematica)
- Il legame tra nome e valore si instaura e si modifica con l'**operatore di assegnamento =**

- In Python, il simbolo `=` ha un significato diverso da “uguaglianza”
- `=` si legge da destra a sinistra. Ad esempio in `x = 3+2`
 - valuto (cioè calcolo il valore) di ciò che sta a destra dell'uguale (in questo caso 5)
 - associo quel valore (5) al nome `x`
 - se al nome `x` era già legato un valore, il precedente legame viene perduto
- Dunque `x = x + 1` ha perfettamente senso!
- Mentre `1 = x` no!
(perché 1 non è un nome legale per un identificatore)
L'operatore `=` **non è commutativo**.

Che cosa stampa questo programma?

```
1 a = 42
2 b = a
3 a = 13
4 print(a, b)
```

Se non siete sicuri, bisogna fare **tracing**, cioè eseguire passo passo a mente/mano il codice.

Oppure farsi aiutare da

<http://pythontutor.com/visualize.html#mode=edit>

Che cosa stampa questo programma?

```
1 a = a + 5  
2 print(a)
```

sintassi

- un'espressione **non è scritta correttamente** (Python non capisce la sequenza e non sa come interpretare il comando),
- non riesce ad eseguire il comando
- es.: `print("Ciao`

runtime

- un **errore eccezionale** del programma,
- l'esecuzione del programma viene interrotta,
- es.: `print(3/0)`

semantica

- il programma scritto **non è quello che si pensa**,
- l'esecuzione può terminare normalmente,
- es.: `print("2 * 5 =", 2+5)`

È quel processo nella produzione del software che ha come obiettivo l'eliminazione degli errori. Durante questo processo è necessario provare (testare) il programma che si è scritto più volte. Per ogni tipo di errore vediamo la sua correzione:

- **sintassi** → utilizzando i messaggi di errore dell'interprete o del compilatore correggere il codice
- **run-time** → prevedere e gestire gli errori che possono verificarsi a run-time per evitare l'interruzione anomala del programma
- **semantica** → effettuare diverse prove, su diversi input, specialmente nei *casi limite*, per capire se l'algoritmo è corretto

Scrivere un programma che scambia il valore di due variabili (indipendentemente dal valore) (es. se inizialmente $a = 7$ e $b = 20$, alla fine `print(a,b)` stamperà `20 7`)

Funzioni

Definizione di funzione

```
1 def funzione(parametri):  
2     istruzione  
3     istruzione  
4     istruzione  
5     ...  
6     istruzione  
7     return risultati
```

Definizione di funzione

- Una funzione ha un nome.
- Una funzione può avere o no dei parametri (ma le parentesi servono sempre!) es.: `def funzione()`
- Una funzione ha un contenuto (il suo *corpo*).
 - Il contenuto va “indentato”, cioè va spostato a destra rispetto alla riga di definizione, per far capire a Python che esso sta “dentro” la funzione. Si possono usare **quattro spazi** oppure un **tab**.
- Una funzione può restituire uno o più risultati (o anche nessuno...) es.: `return`

Definizione di funzione

- Proprio come in matematica, definisco la funzione una volta sola (es. $f(x) \stackrel{\text{def}}{=} x^2$)

- In Python:

```
1 def f(x):  
2     return x**2
```

- Poi però posso usarla quante volte voglio (es. $f(3)$ vale 9, $f(a)$ è uguale ad a^2 , eccetera)...
- Nella shell di Python:

```
>>> f(3)  
9
```

```
1 def funzione(parametri):  
2     """commento che spiega cosa fa la funzione"""  
3     istruzione  
4     istruzione #commento breve  
5     istruzione  
6     ...  
7     istruzione  
8     return risultati
```

- Sono leggibili dagli esseri umani e ignorati dal computer
- Servono a documentare il codice
- Buona norma: subito sotto la riga in cui dichiariamo la funzione, mettere un commento (incluso tra due triplette di doppi apici) che spiega sintenticamente cosa fa la funzione
- Accanto a istruzioni **complesse**, mettere un commento breve (che inizia con #) per spiegarle

Importazione di moduli

```
1 import modulo
2 def funzione(parametri):
3     """commento che spiega cosa fa la funzione"""
4     istruzione
5     istruzione #commento breve
6     modulo.funzione()
7     ...
8     istruzione
9     return risultati
```


Import modulo

Comando per richiamare funzioni implementate in un file salvato in precedenza. Esempi di moduli già presenti in Python sono:

- **math**:
 - sono implementate funzioni matematiche ad esempio: `sin(x)`, `cos(x)`, `sqrt(x)`, ...
 - sono definite le costanti `e` (numero di Nepero), `pi` (π)
- **random**:
 - sono implementate alcune funzioni per la generazione di numeri pseudocasuali, es: `randint(a,b)`

Import modulo

Comando per richiamare funzioni implementate in un file salvato in precedenza. Esempi di moduli già presenti in Python sono:

- **math**:
 - sono implementate funzioni matematiche ad esempio: `sin(x)`, `cos(x)`, `sqrt(x)`, ...
 - sono definite le costanti **e** (numero di Nepero), **pi** (π)
- **random**:
 - sono implementate alcune funzioni per la generazione di numeri pseudocasuali, es: `randint(a,b)`
- Per usare in un programma una o più funzioni appartenenti a un modulo:
 - importare il modulo prima del suo utilizzo es: `import math`
 - richiamare la funzione in riferimento al modulo es.
`math.sin(90)`
- Per sapere cosa c'è dentro una libreria, consultare la documentazione:

<https://docs.python.org/3/library/math.html>

1. Cosa fa questa funzione? Aggiungere il commento di spiegazione
2. Rinominare la funzione con un nome significativo
3. Provare ad eseguire il programma. Che succede?

```
1 import math
2 def funzione(a,b):
3     c = a**2+b**2
4     d = math.sqrt(c)
5     return d
```

Pair programming: cambio!

Esercizio da fare in classe

Scrivere una funzione `secondiInOreMinSec` che prende come parametro un intero non negativo che rappresenta i secondi e restituisce a quante ore, minuti, secondi corrispondono.

Esempio: 4000 secondi corrispondono a 1h 6m e 40s. Dunque nell'esempio, se chiamo la funzione con parametro 4000, otterrò:

```
>>> secondiInOreMinSec(4000)
(1, 6, 40)
```

Esercizio da fare in classe

Suggerimento: serviranno divisioni intere e resti con valori quali 3600, 60...

In generale è bene prevedere il comportamento e poi testare alcuni casi limite:

- `secondiInOreMinSec(0)` \rightarrow (0,0,0)
- `secondiInOreMinSec(60)` \rightarrow (0,1,0)
- `secondiInOreMinSec(125)` \rightarrow (0,2,5)
- `secondiInOreMinSec(3600)` \rightarrow (1,0,0)
- `secondiInOreMinSec(3661)` \rightarrow (1,1,1)

Esercizi per casa

Esercizi per casa

1. Scrivere una funzione che non ha nessun parametro, **non restituisce nulla**, ma stampa a video il valore (approssimato) di \sqrt{e} (radice quadrata del numero di Nepero).
2. Sia C il capitale iniziale di un investimento. Sia r il tasso di interesse (espresso come decimale, es 0.03), sia n il numero di volte che gli interessi vengono calcolati ogni anno e sia t il numero di anni. Il capitale finale M si calcola allora come:

$$M = C \left(1 + \frac{r}{n}\right)^{nt}$$

Scrivere una funzione che ha come parametri C, r, n, t e **restituisce** il valore di M , ma **non stampa nulla**.

Nello stesso file scrivere poi un esempio che, **usando la funzione**, stampa: **Capitale finale per investimento di 10.000, calcolo mensile, tasso 8%, per 2 anni: 11728.879317453097**