Lezione 2 Python

Stile del codice:

1. Una istruzione per riga, non è necessario l'uso del ';'.
2. Si può spezzare il comando con il carattere '\', mentre il carattere ';' può essere usato per concatenare due istruzioni sulla stessa riga (sconsigliato).
3. Le istruzioni contenute nelle parentesi (), [], {} non necessitano del carattere '\' per continuare suller righe successive.
4. Per raggruppare un blocco di istruzioni si usa l'indentazione: consigliato l'uso di 4 spazi (non del tab). *Nota: mescolare spazi e tab da errore da Python3 in poi.*
5. Le virgolette, o apici, servono per definire stringhe di caratteri. Python accetta indistintamente virgolette singole ('') o doppie (""). Inoltre le virgolette triple (''' o """) vengono utilizzate per definire una stringa su più righe.
6. I commenti su una riga iniziano col simbolo '#'.
7. Linee guida per la scrittura di codice python definite nel PEP 8 - Style Guide for Python Code.
8. Lo stile più comune prevede un massimo di 79 caratteri per linea di codice, per rendere più fruibile la lettura.

Esempio:

```
def concatena(parola, n):
    """
    Questa funzione concatena la stringa parola n volte.
    La funzione è ricorsiva.
    """
    if n <= 1:
        return parola
    else:
        # chiamata ricorsiva
        return parola + ", " + concatena(parola, n - 1)

s = concatena('ciao', 3)
print(s)
```

Output:

```
>>> ciao, ciao, ciao
```

Tipi di dato

Python è un linguaggio dinamicamente tipizzato.

Text Type	str (stringa)
Numeric Types	int, float, complex
Sequence Types	list, tuple, range
Mapping Types	dict (dizionari)
Set Types	set, frozenset
Boolean Type	bool
Binary Types	bytes, bytearray, memoryview

Esercizio sui Tipi: La funzione Type()

La funzione `Type()` permette di vedere il tipo dell'oggetto:

```
# Esempio
print(type(5))                                  # >>> <class 'int'>
print(type(10/2))                               # >>> <class 'float'>
print(type(10//2))                              # >>> <class 'int'>
print(type(100000000000000000000000))           # >>> <class 'int'>
print(type('ciao'))                             # >>> <class 'str'>
print(type(print(5)))                           # >>> <class 'NoneType'>
print(type([1, 2, 3]))                          # >>> <class 'list'>
# radice di 2
print(type(2**(1/2)))                           # >>> <class 'float'>
# radice di -2
print(type((-2)**(1/2)))                        # >>> <class 'complex'>
```

Dichiarazioni di Variabili:

"Una delle caratteristiche più potenti di un linguaggio di programmazione è la capacità di elaborare delle variabili"

1. Una variabile è un nome che fa riferimento ad un oggetto che contiene un valore.
2. **Attenzione**: Una variabile non è il nome simbolico di un valore, come avviene ad esempio in C.
3. Un nome valido di variabile:
   3.1 può contenere solo lettere, numeri e il carattere '_'
   3.2 non può contenere spazi
   3.3 non può iniziare con un numero
   3.4 non deve essere una **parola chiave** del linguaggio
   3.5 i nomi delle variabili sono *case-sensitive*
   3.6 può contenere caratteri non ASCII (utf-8) (sconsigliato!)

Convenzioni:

Ci sono delle convenzioni relative alla denominazione degli identificatori di Python:

- I nomi delle classi iniziano con una lettera maiuscola, tutti gli altri identificatori iniziano con una lettera minuscola
- Un identificatore che inizi con *un trattino basso* indica che si tratta di un *identificatore privato*

```
# Identificatore privato
_identif_privato
```

- Un identificatore che inizi con **due trattini bassi** indica che si tratta di un **identificatore fortemente privato**

```
# Identificatore fortemente privato
__identif_fort_privato
```

- Se l'identificatore inizia e finisce con due trattini bassi allora è un nome speciale definito nel linguaggio

```
# Identificatore speciale
__identif_speciale__
```

Parole chiave del linguaggio:

Keywords	in	Python	
False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	whit
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Variabili e Istruzioni di Assegnamento

Un'istruzione di assegnamento serve a creare una nuova variabile, specificandone il nome, e ad assegnarle un valore (non indichiamo esplicitamente il tipo). Il carattere '=' associa ad un nome un riferimento ad un oggetto chediventa raggiunggibile.

```
# Esempio
a = 10
b = 5
```

L'operatore `is` confronta l'identità di due oggetti; la funzione `id()` restituisce un intero che rappresenta l'identità dell'oggetto.
`hex()` converte un valore numerico in esadecimale, utile per stampare degli indirizzi di memoria.

Esempi:

```
# Creaiamo due variabili e assegnamo un valore
a = 5
b = 10
# Stampiamo l'indirizzo di memoria delle variabili, in esadecimale
print(hex(id(a)), hex(id(b)))
```

Output:
`>>> 0x955ec0 0x955f60`
Gli indirizzi di memoria sono diversi

```
# Se ora modifichiamo b e la rendiamo uguale ad a
b = 5
# Ora le due variabili puntano allo stesso oggetto '5'
print(hex(id(a)), hex(id(b)))
```

Output:
`>>> 0x955ec0 0x955ec0`
Gli indirizzi di memoria sono uguali
a e **b** sono oggetti che puntano rispettivamente a 5 e 10. Dopo l'assegnamento `b = 5`, entrambi puntano allo stesso oggetto.

Espressioni matematiche e logiche: priorità degli operatori

Quando un'espressione contiene più operatori, la successione con cui viene eseguito il calcolo dipende dallàordine delle operazioni. Per quelle matematiche Python segue la stesse regole di precedenza comunemente usate in matematica. L'acronimo **PEMDAS** è molto utile per ricordare le regole: **P**arentesi, **E**levamento a Potenza, **M**oltiplicazione e **D**ivisione, **A**ddizione e **S**ottrazione.

Usando *sempre* le parentesi, si evita di commettere errori!

Esercizio:

Calcolare il volume di una sfera

```
import math as mt
r = 10
volume = (4/3) * mt.pi * r**3
print(volume)
```

Output:

```
>>> 4188.790204786391
```

Espressioni con stringhe

Operatore	Operazione	Esempio	Valutata in ...
+	concatenamento	'Hello ' + 'World'	'Hello World'
*	ripetizione	'Hello' * 2	'HelloHello'

```
# Esempio
print('-' * 20 + '|')
```

Output:

```
>>> --------------------|
```

Le Funzioni

Una funzione è una serie di istruzioni che esegue un calcolo, alla quale viene assegnato un nome. Per definire una funzione, si deve specificarne il nome e scrivere la serie di istruzioni. In un secondo tempo, è possibile "chiamare" la funzione mediante il nome assegnato.

```
# Esempio di funzione che stampa l'indirizzo di memoria di un parametro:
def stampa_memoria(x):
    print(hex(id(x)))

# Esempio di funzione che stampa due volte il parametro
def stampa_due_volte(x):
    print(x*2)

stampa_due_volte('ciao')
stampa_memoria('ciao')
```

Output:

```
>>> ciaociao
>>> 0x7f9417333d30
```

Funzioni: scopo variabili

Una variabile dichiarata dentro ad una funzione si può usare ('vedere') solo al suo interno. Si dice che questa variabile è **locale**.

```
# Esempio di variabile locale
def concatena(x, y):
    cat = x + ' ' + y
    print(cat)

x = 'ciao'
y = 'mondo'
concatena(x, y)

# ERRORE: non posso accedere a cat
print(cat)
```

Output:

```
>>> ciao mondo
```

```
---------------------------------------------------------------------------
NameError Traceback (most recent call last)
Input In [3], in <cell line: 10>()
8 concatena(x, y)
9 # ERRORE! non posso accedere a cat
---> 10 print(cat)
NameError: name 'cat' is not defined
```

*La variabile **cat** è locale, non può essere stampata*

Notare che **x** e **y** nella funzione NON sono la stessa cosa di **x** e **y** fuori (parametri formali vs parametri attuali). Se si vuole usare una variabile esterna dentro a una funzione, senza passarla come parametro, questa deve essere **globale**. Tutte le variabili create fuori dalle funzioni, sono dette variabili globali. Le possiamo leggere, ma NON modificare (per quello serve la parola chiave ***global***).

```
x = 'ciao'
def modifica_locale():
    x = 10

def modifica_globale():
    global x
    x = 10

# La prima funzione non modifica il valore di 'x'
modifica_locale()
print(x)
# Mentre la seconda si
modifica_globale()
print(x)
```

Output:

```
>>> ciao
>>> 10
```

Liste

Le liste sono **sequenze di valori**. Possono contenere elementi eterogenei (e.g., possono combinare stringhe e interi).

Esempio:

```
terna = [23, 72, 8]
primari = ['giallo', 'magenta', 'ciano']
voti = [27, 30, 24, 'idoneo']
```

Le liste sono un primo esempio di **variabili mutabili**.

Esempi di variabili mutabili sono:

- liste
- dizionari
- set
- oggetti

Tra i tipi immutabili abbiamo:

- numeri
- stringhe e tuple
- frozenset

Una lista punta ad un elemento che posso mutare. Se più cose puntano allo stesso elemento, le modifiche vengono viste da tutte le variabili che puntano allo stesso elemento (mutabile).

Esempio:

```
a = 5
b = 10
c = a
print(hex(id(a)), hex(id(b)), hex(id(c)))
a = 10
print(hex(id(a)), hex(id(b)), hex(id(c)))
```

Output:

```
>>> 0x7fb704d889b0 0x7fb704d88a50 0x7fb704d889b0
>>> 0x7fb704d88a50 0x7fb704d88a50 0x7fb704d889b0
```

Mentre per le liste le cose cambiano:

```
l = [1, 2, 3]
h = l
print(l, h)
l.append(4)
# Cambia anche h!
print(l, h)
```

Output:

```
>>> [1, 2, 3] [1, 2, 3]
>>> [1, 2, 3, 4] [1, 2, 3, 4]
```

Per modificare solo `l` e mantenere `h` inalterata posso usare l'operatore `+`

```
l = l + [5]
print(l, h)
```

Output:

```
>>> [1, 2, 3, 4, 5] [1, 2, 3, 4]
```

Inoltre posso creare una lista di n elementi con l'operatore `*` nel seguente modo:

```
# creo una lista di 10 elementi tutti a 0.
k = [0] * 10
print(k)
```

Output:

```
>>> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Le liste possono contenere oggetti di vario tipo, anche diversi tra loro:

```
j = [1, 'ciaio', [1, 2, 3]]
print(j)
```

Output:

```
>>> [1, 'ciao', [1, 2, 3]]
```

Infine, posso creare una lista di liste nel seguente modo:

```
init = [ [] ] * 10
init[0].append(1)
print(init)
```

Output:

```
>>> [[1], [1], [1], [1], [1], [1], [1], [1], [1], [1]]
```

Attenzione: *siccome tutte le liste all'interno di* `init` *puntano allo stesso oggetto, quindi modificandone il primo elemento, vengono modificati tutti.*

Liste: creare copie

Se si vuole modificare una lista mantenendo anche l'originale, si deve crearne una copia. Un modo per farlo è utilizzare l'operatore `+`.

```
# Esempio
l = [1, 2, 3]
h = l
print(l, h)

# Ora la lista h non avrà il quarto elemento
l = l + [4]
print(l, h)
```

Output:

```
>>> [1, 2, 3] [1, 2, 3]
>>> [1, 2, 3, 4] [1, 2, 3]
```

Esercizio

Data una lista di 10000 elementi, misurare quanto tempo ci vuole per riempirla

```
import time

start = time.time()

l1 = []
for i in range(10000):
    # Aggiungo l'elemento i-esimo alla lista ad ogni iterazione
    l1.append(i)

end = time.time()

print(len(l1), end - start)
```

Output:

```
>>> 10000 0.0008478164672851562
```

Applichiamo lo stesso concetto, ma con l'operatore `+`

```
import time

start = time.time()
l2 = []
for i in range(10000):
    # Ricopia la lista ad ogni iterazione, aggiungendo un nuovo elemento i-esimo
    l2 = l2 + [i]

end = time.time()

print(len(l2), end - start)
```

Output:

```
>>> 10000 0.1261458396911621
```

Il secondo caso è molto più lento!

Liste come argomenti di funzioni

Se viene passata una lista come argomento di funzione, questo crea un nuovo puntatore allo stesso elemento. Un `append` modifichera' la lista originale. Se si vuole gestire la lista senza cambiarla, serve una copia.

```
# Modifica la lista originale
def man_list(l):
l.append(0)
print(l)
# Modifica una copia della lista senza modificare l'originale
def man_list_alt(l):
l = l + [0]
print(l)
l = ['python']
man_list(l)
man_list(l)
man_list(l)
man_list_alt(l)
man_list_alt(l)
man_list_alt(l)
# la lista e' rimasta quella originale all'esterno della funzione
print(l)
```

Outputs:

```
['python', 0]
['python', 0, 0]
['python', 0, 0, 0]
['python', 0, 0, 0, 0]
['python', 0, 0, 0, 0]
['python', 0, 0, 0, 0]
['python', 0, 0, 0]
```

Esercizio PythonTurtle

Esercizio capitolo 4 del libro Think Python

1. Write a function called sqare that takes a pramaetre named t, which is a turtle. It should use the turtle to draw a square. Write a function call that passes bob as an argument to square, then return the program again
2. Add another parameter, named length, to square. Modify the body so length of the sides is length, and then modify the function call to provide a second argument. Run the program again. Test your program with a range of valuse for length
3. The functions lt and rt make 90-degree turns by default, but you can provide a second argument that specifies the number of degrees. For example, lt(bob, 45) turns bob 45 degrees to the left. Make a copy of sqare and change the name to polygon. Add another parameter named n and modify the body so it draws a n-sided regular polygon. Hint: The exterior angles of an n-sided regular polygon are 360/n degrees.
4. Write a function circle that takes a turtle, t and radius, r, as parameters and that draws an approximate circle by invoking polygon with an appropriate length and number of sides. Test your functin with a range of values of r.

```
import turtle
import math

# 1 & 2:
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)
# 3:
def polygon(t, length, n):
    for i in range(n):
        t.fd(length)
        t.lt(360/n)
# 4:
def circle(t, radius):
    length = (2 * math.pi * radius) / 360
    polygon(t, length, 360)

bob = turtle.Turtle()
square(bob, 100)
polygon(bob, 100, 6)
bob.fd(50)
bob.rt(90)
bob.fd(12)
bob.lt(90)
circle(bob, 100)
turtle.mainloop()
```