

4 Prima lezione

4.1 Funzione print

Se un macchina fosse senziente e gentile, quindi non un'intelligenza artificiale cresciuta su twitter, forse come prima cosa salterebbe tutti e il modo per comunicare è la funzione print, vediamo quindi il più classico degli esempi:

```
1 print('Hello world!')
2
3 [Output]
4 Hello world!
```

Questa funzione può stampare sia valori che espressioni:

```
1 print('Hello world!')
2 print('42')
3
4 print('Hello world!', 42)
5 print('Adesso vado \n a capo')
6
7 [Output]
8 Hello world!
9 42
10 Hello world! 42
11 Adesso vado
12 a capo
```

4.2 Commenti

Al fine di rendere fruibile ad altri o anche al voi stesso del futuro il codice è opportuno inserire i commenti, ovvero frasi che non vengono lette dall'interprete (o dal compilatore) che spiegano cosa voi stiate facendo; altrimenti vi ritroverete nella scomoda situazione in cui solo Dio saprebbe spiegarvi il funzionamento del vostro codice.

```
1 #per i commenti che occupano una singola linea di codice si usa il cancelletto
2 #stampo hello world
3 print('Hello world!')
4
5 """
6 per un commento di maggiori linee di codice
7
8 vanno usate tre virgole per racchiuderlo
9 """
10 '''
11 ma van bene
12
13 anche tre apici
14 '''
15 [Output]
16 Hello world!
```

4.3 Variabili

Una variabile è un nome, un simbolo, che si dà ad un certo valore. In Python non è necessario né definire le variabili prima di utilizzarle, né specificare il loro tipo, esse si creano usando il comando di assegnazione '='. Facciamo un esempio con variabile numeriche:

```
1 numerointero= 13
2 numeroavirgolamobile= 13.
3
4 print('Numero intero:', numerointero, 'Numero in virgola mobile:', numeroavirgolamobile)
5 #oppure possiamo stampare in questo modo:
6 print(f'Numero intero: {numerointero}, Numero in virgola mobile: {numeroavirgolamobile}')
7
8 [Output]
9 Numero intero: 13 Numero in virgola mobile: 13.0
10 Numero intero: 13 Numero in virgola mobile: 13.0
```

Ovviamente le variabili possono essere non solo numeri ma anche molto altro, e possiamo verificarne il tipo grazie alla funzione 'type()':

```
1 #inizializziamo delle variabili
2 n = 7
```

```

3 x = 7.
4 stringa = 'kebab'
5 lista = [1, 2., 'cane']
6 tupla = (42, 'balena')
7 dizionario = {'calza': 0, 'stampante': 0.5}
8
9 #stampiamole e stampiamone il tipo
10 print(n, type(n))
11 print(x, type(x))
12 print(stringa, type(stringa))
13 print(lista, type(lista))
14 print(tupla, type(tupla))
15 print(dizionario, type(dizionario))
16
17 [Output]
18 7 <class 'int'>
19 7.0 <class 'float'>
20 kebab <class 'str'>
21 [1, 2.0, 'cane'] <class 'list'>
22 (42, 'balena') <class 'tuple'>
23 {'calza': 0, 'stampante': 0.5} <class 'dict'>

```

Dizionari, liste, tuple, possono essere elementi molto utili. Tutti essi sono indicizzati, i primi due sono modificabili le tuple invece sono immutabili, come abbiamo visto non ci sono particolari problemi di casting, essi possono contenere elementi di vario genere. I dizionari inoltre, utilizzando un sistema chiave valore possono essere utili per gestire alcuni output di codici lunghi o complessi; per esempio sono molto utili nella programmazione in parallelo, dove l'ordine si perde quindi un dizionario è un ottimo modo per tenere traccia di tutta l'esecuzione. Vediamo ora come fare le classiche operazioni matematiche:

```

1 x1 = 3
2 y1 = 4
3
4 somma = x1 + y1
5 prodotto = x1*y1
6 differenza = x1 - y1
7 rapporto = x1/y1
8 potenza = x1**y1
9
10 print(somma, prodotto, differenza, rapporto, potenza)
11 [Output]
12 7 12 -1 0.75 81

```

Vale la pena dilungarsi un attimo su una piccola questione: l'aritmetica in virgola mobile è intrinsecamente sbagliata poiché giustamente il computer ha uno spazio di memoria finita e quindi non può tenere infinite cifre decimali:

```

1 x = 0.1
2 y = 0.2
3 z = 0.3
4
5 print(x + y)
6 print(z)
7
8 [Output]
9 0.30000000000000004
10 0.3

```

Il precedente è solo uno tra i molti esempi che si potrebbero fare per far notare come l'aritmetica dei numeri in virgola mobile possa dare problemi; Il lettore provi a vedere se è vero che $a + (b + c) = (a + b) + c$ con a, b, c numeri in virgola mobile, probabilmente il computer non sarà d'accordo. Ai computer i numeri in virgola mobile, i numeri reali, non piacciono molto, preferiscono i numeri interi e quelli razionali (e ovviamente adorano le potenze di due, grazie codice binario):

```

1 #variabili
2 x = 0.1
3 y = 0.2
4 z = 0.3
5 #sommo le prime due
6 t = x + y
7
8 """
9 applico una funzione che mi fornisce
10 una tupla contenente due numeri interi
11 il cui rapporto restituisce il numero iniziale.
12 output del tipo: (numeratore, denominatore)

```

```

13 """
14 print(t.as_integer_ratio())
15 print(z.as_integer_ratio())
16
17 [Output]
18 (1351079888211149, 4503599627370496)
19 (5404319552844595, 18014398509481984)

```

Per quanto riguarda i numeri in virgola mobile, possiamo scegliere quante cifre dopo la virgola stampare, vediamo con un esempio:

```

1 #definiamo una variabile
2 c = 3.141592653589793
3
4 #stampa come intero
5 print('%d' %c)
6
7 #stampa come reale
8 print('%f' %c) #di default stampa solo prime 6 cifre
9 print(f'{c}') #di default stampa tutte le cifre
10
11 #per scegliere il numero di cifre, ad esempio sette cifre
12 print('%.7f' %c)
13 print(f'{c:.7f}')
14
15 [Output]
16 3
17 3.141593
18 3.141592653589793
19 3.1415927
20 3.1415927

```

Notare che il computer esegue un arrotondamento.

Una variabile può essere ridefinita e cambiare valore, il computer userà l'assegnazione più recente:

```

1 #definiamo una variabile
2 x = 30
3
4 """
5
6 operazioni varie
7
8
9 """
10
11 #ridefiniamo la variabile
12 x = 18
13
14 print('x=', x)
15
16 """
17 E' possibile anche sovrascrivere una variabile
18 con un numero che dipende dal suo valore precedente:
19 """
20 x = x + 1 #incrementiamo di uno
21 #oppure:
22 x += 1
23 print('x=', x)
24
25 x = x * 2 #moltiplichiamo per due
26 #oppure:
27 x *= 2
28 print('x=', x)
29
30 [Output]
31 x= 18
32 x= 20
33 x= 80

```

Come si vede i vari comandi $x = x \text{ operazione numero}$ possono essere abbreviati con $x \text{ operazione} = \text{numero}$.

4.4 Librerie

Le librerie sono luoghi mistici create dagli sviluppatori, esse contengono molte funzioni, costanti e strutture dati predefinite; in generale se volete fare qualcosa esisterà una libreria con una funzione che implementa quel qualcosa o che comunque vi può aiutare in maniera non indifferente. Prima di poter accedere ai contenuti di una

libreria, è necessario importarla. Per farlo, si usa il comando `import`. Solitamente è buona abitudine importare tutte le librerie che servono all'inizio del file. Ecco un paio di esempi:

```
1 import numpy
2
3 #per usare un contenuto di questa libreria basta scrivere numpy.contenuto
4 pigreco = numpy.pi
5 print(pigreco)
6
7 #Possiamo anche ribattezzare le librerie in questo modo
8 import numpy as np
9 #da ora all'interno del codice numpy si chiama np
10
11 eulero = np.e
12 print(eulero)
13
14 [Output]
15 3.141592653589793
16 2.718281828459045
```



```
1 import math
2
3 coseno=math.cos(0)
4 seno = math.sin(np.pi/2) #python usa di default gli angoli in radianti!!!
5 senosbagliato = math.sin(90)
6
7 print('Coseno di 0=', coseno, "\nSeno di pi/2=", seno, "\nSeno di 90=", senosbagliato)
8
9 #bisogna quindi stare attenti ad avere tutti gli angoli in radianti
10 angoloingradi = 45
11 #questa funzione converte gli angoli da gradi a radianti
12 angoloinradianti = math.radians(angoloingradi)
13
14 print("Angolo in gradi:", angoloingradi, "Angolo in radianti:", angoloinradianti)
15
16 [Output]
17 Coseno di 0= 1.0
18 Seno di pi/2= 1.0
19 Seno di 90= 0.8939966636005579
20 Angolo in gradi: 45 Angolo in radianti: 0.7853981633974483
```

Le due librerie qui usate contengono funzioni simili, ad esempio il seno è implementato sia in `numpy` che in `math`, cambia il fatto che `math` può calcolare il seno di un solo valore, mentre `numpy`, come vedremo, può calcolare il seno di una sequenza di elementi. Come sapere tutte le possibili funzioni contenute nelle librerie e come usarle? "Leggetevi il cazzo di manga" (n.d.r. Leggetevi la documentazione disponibile tranquillamente online)

4.5 Gestione errori

È molto facile scrivere codice che produca errore, magari perchè distrattamente ci siamo dimenticati qualcosa o magari qualcosa è stato implementato male. Esiste un costrutto che ci permette di gestire gli errori in maniera tranquilla diciamo. Facciamo un semplice esempio:

```
1 a = 0
2 b = 1/a
3 print(b)
4
5 [Output]
6 Traceback (most recent call last):
7   File "<tmp 1>", line 5, in <module>
8     b = 1/a
9 ZeroDivisionError: division by zero
```

Abbiamo fatto una cosa molto brutta, nemmeno Dio può dividere per zero (cosa non proprio vera infatti si hanno ancora teorie fisiche con divergenze purtroppo) e quindi il computer ci da errore. Possiamo aggirare il problema, evitando così il second impact, in due modi diciamo:

4.5.1 Try e except

Possiamo utilizzare il costrutto `try except` dicendo al computer: prova a fare la divisione e, se questa da errore, e l'errore è "ZeroDivisionError" allora assegna a `b` un altro valore. in questo modo eventuali istruzioni presenti dopo vengono eseguite e il codice non si arresta.

```
1 a = 0
2 try :
```

```

3     b = 1/a
4 except ZeroDivisionError:
5     b = 1
6
7 print(b)
8
9 [Output]
10 1

```

4.5.2 Raise Exception

Mettiamo il caso in cui ci siano operazioni da fare in cui il valore della variabile "b" è importante, quindi sarebbe meglio interrompere il flusso del codice perché con un dato valore il risultato finale sarebbe poco sensato. Si può fare il controllo del valore e sollevare un'eccezione per fermare il codice.

```

1 """
2 leggo un valore da shell
3 uso del comando try per evitare che venga letto
4 qualcosa che non sia un numero: e.g. una stringa
5 """
6 try:
7     b = int(input('scegliere un valore:'))
8 except ValueError:
9     print('hai digitato qualcosa diverso da un numero, per favore ridigitare')
10    b = int(input('scegliere un valore:'))
11
12 #se si sbaglia a digitare di nuovo il codice si arresta per ValueError
13
14 #controllo se e' possibile proseguire
15 if b > 7 :
16     #se vero si blocca il codice sollevando l'eccezione
17     messaggio_di_errore = 'il valore scelto risulta insensato in quanto nulla supera 7, misura
18     massima di ogni cosa'
19     raise Exception(messaggio_di_errore)
20
21 [Output]
22 8
23 Traceback (most recent call last):
24   File "<tmp 1>", line 18, in <module>
25     raise Exception(messaggio_di_errore)
26 Exception: il valore scelto risulta insensato in quanto nulla supera 7, misura massima di ogni
27     cosa

```

Non abbiamo ancora spiegato cosa è un if ma dal codice direi che non risulta di difficile intuizione.

4.6 Come leggere il Traceback

Quelli che abbiamo sopra sono esempi di errori che danno come risultato in genere l'interruzione del codice. Quindi quando la shell di pyzo si tinge di rosso cremisi che nemmeno fosse appena passata una battaglia campale di Vlad figlio del drago voivoda di Valacchia è buona norma leggere attentamente il messaggio di errore, il traceback (che come suggerisce il nome va letto al contrario; da sotto verso sopra), e capire cosa si è sbagliato. Il traceback infatti è la catena di eventi che hanno portato all'errore. Analizziamo il caso di sopra.

```

1 Traceback (most recent call last):
2   File "<tmp 1>", line 5, in <module>
3     b = 1/a
4 ZeroDivisionError: division by zero

```

La prima riga ci fa capire che c'è stato un errore.

La seconda ci dice che nel file chiamato " < tmp1 > " (perché mi ero dimenticato di salvarlo ancora, sennò ci sarebbe il path del file) alla linea 5, nel codice eseguito, se fosse dentro una funzione ci sarebbe, oltre a questa, un' altra riga uguale con il nome della funzione al posto di < module >, è successo qualcosa.

La terza linea è la linea di codice a cui è avvenuto il misfatto.

La quarta ci da due informazioni, separate dai due punti: il tipo di errore che è avvenuto e le informazioni riguardo ad esso.

Facciamo adesso un esempio un po' più complicato, in cui useremo funzioni quindi magari potete tornare a rileggerlo dopo se volete.

```

1 def err(c):
2     b = 1/c
3     return b
4

```

```

5 def run(c):
6     print(err(c))
7
8 if __name__ == "__main__":
9     run(0)
10
11 [Output]
12 Traceback (most recent call last):
13   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 9, in
    <module>
14     run(0)
15   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 6, in
    run
16     print(err(c))
17   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 2, in
    err
18     b = 1/c
19 ZeroDivisionError: division by zero

```

Leggiamolo insieme, partendo dal basso abbiamo:

C'è stata una divisione per zero che ha causato l'errore di tipo ZeroDivision error alla linea 2 nella funzione "err" contenuta nel codice avente quel path (sta volta il codice era salvato).

L'errore è stato causato dalla chiamata della funzione "err" da parte della funzione "run" a linea 6, contenuta nello stesso file, hanno infatti stesso path.

Ma ancora prima l'errore è causato dalla chiamata della funzione "run" (che ha chiamato err, che ha prodotto l'errore) all'interno dello stesso codice, alla linea 9. Insomma fiera dell'est di Branduardi spostati proprio.