4 The Zen of Python

Una volta installato pyzo, oppure python, aprite un terminale, che sia quello di pyzo, la shell di ubuntu (dopo aver digitato python3), o di anaconda, e scrivete:

```
>>> import this
```

ecco cosa uscirà:

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

5 Prima lezione

5.1 Funzione print

Se un macchina fosse senziente e gentile, quindi non un'intelligenza artificiale cresciuta su twitter, forse come prima cosa saluterebbe tutti e il modo per comunicare è la funzione print, che ci permette di stampare a schermo (sulla shell) delle informazione contenute nel codice. Vediamo quindi il più classico degli esempi:

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

Bene, se siete riuscire ad eseguire questo codice siete ufficialmente dei programmatori. Giusto per voler essere pedanti i numeri che vedete a sinistra non hanno un vero e proprio siginificato per l'esecuzione; il loro unico scopo è indicare all' utente a che riga si trova. Questa funzione può stampare sia valori che espressioni (in gergo stringhe) [capiremo tra poco cosa queste magiche cose che vengono stampate sono effettivamente]:

```
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
```

5.2 Commenti

Come vedete dal codice precedente alla linea 4 ci sta un simbolo poco familiare a chi per la prima volta approccia la programmazione: "\n" chi era costui? Scritto così il codice, l'unico modo per capirlo è che voi eseguiate il codice e vedendo il risultato cerchiate di risalire al significato. Ora chiaramente questa procedura è abbastanza sbrigativa ma se dovete capire diverse parti del codice, il quale magari impiega un tempo non trascurabile a darvi un risultato, beh diciamo che non è una bella vita. Al fine dunque di rendere fruibile sia 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 una 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!
```

Ricordate, un codice viene letto molte più volte rispetto a quanto viene scritto. Quindi commentate, sempre.

5.3 Variabili

Una variabile è un nome, un simbolo, che si da ad un certo valore. In Python non è necessario definire le variabili prima di utilizzarle specificandone il tipo, come faremmo in C o fortran, esse si creano, o meglio si inizzializzano, 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:
```

```
print(f'Numero intero: {numerointero}, Numero in virgola mobile: {numeroavirgolamobile}')

[Output]
Numero intero: 13 Numero in virgola mobile: 13.0
Numero intero: 13 Numero in virgola mobile: 13.0
```

Un' altra cosa molto fondamentale, oltre i commenti, per la fuibilità del codice è il modo di dare nomi alle variabili. Nel codice di sopra il nome delle variabili è alquanto esplicativo del loro significato, ed è in genere buona norma, appunto, dare nomi che siano intuibili. Ora non vi dico che dovete chiamare una variabile: "momentoagolaresullasseZ", che ci vogliono tre anni solo a scriverla, che nel frattempo pure il protone inizia a decadere, ma di certo chiamarla "L_z" piuttosto che "a" o "pippo" è una strada che andrebbe perseguita. Ovviamente le variabili possono essere non solo numeri ma anche molto altro, e possiamo verificarne il tipo grazie alla funzione 'type()':

```
#inizializziamo delle variabili
n = 7
x = 7.
stringa = 'kebab'
lista = [1, 2., 'cane']
tupla = (42, 'balena')
dizionario = {'calza': 0, 'stampante': 0.5}

#stampiamole e stampiamone il tipo
print(n, type(n))
print(x, type(x))
print(stringa, type(stringa))
print(lista, type(lista))
print(tupla, type(tupla))
print(dizionario, type(dizionario))

[Output]
7 <class 'int'>
7.0 <class 'float'>
kebab <class 'str'>
[1, 2.0, 'cane'] <class 'list'>
(42, 'balena') <class 'tuple'>
{'calza': 0, 'stampante': 0.5} <class 'dict'>
```

Dizionari, liste, tuple, possono essere elementi molto utili. Giusto qualche info un po' anticipata 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 cioè possono contenere elementi di vario genere. I dizionari inoltre, utilizzando un sistema chiave valore possono essere utili per gestire alcuni output di codici lungi 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. Più avanti parleremo un po' di questo altro tipo di variabili. Concentriamoci attualmente su come fare le classiche operazioni matematiche tra variabili, numeriche ovviamete:

```
x1 = 3
y1 = 4

somma = x1 + y1
prodotto = x1*y1
differenza = x1 - y1
rapporto = x1/y1
potenza = x1**y1

print(somma, prodotto, differenza, rapporto, potenza)
[Output]
7 12 -1 0.75 81
```

E finqui tutto bene, tutto abbastanza normale e mi raccomando ricordate che l'elevamento a potenza si fa con doppio asterisco "**". 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. A seconda del tipo della variabile ci si ferma ad tot di cifre decimali, 8 per i float32, 16 per i float64; dove il numero che segue la parola float indica il numero di bit che il computer usa per scrivere il numero. Questa precisione può comunque essere cambiata grazie alla libreria : "mpmath"; la quale permette di settare una precisione arbitraria, divertitevi a scoprirla. Vediamo un classico esempio:

```
x = 0.1
y = 0.2
z = 0.3

print(x + y)
```

```
print(z)

[Output]
0.30000000000000004
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. Come esercizio lasciato al lettore provate 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):

```
#variabili
x = 0.1
y = 0.2
z = 0.3
#sommo le prime due
t = x + y

"""
applico una funzione che mi fornisce
una tupla contenente due numeri interi
il cui rapporto restituisce il numero iniziale.
output del tipo: (numeratore, denominatore)
"""
print(t.as_integer_ratio())
print(z.as_integer_ratio())

[Output]
(1351079888211149, 4503599627370496)
(5404319552844595, 18014398509481984)
```

Vediamo quindi che un numero reale è scritto in realtà, e altrimenti non si potrebbe fare, come numero razionale. Per quanto riguarda i numeri in virgola mobile, possiamo scegliere quante cifre dopo la virgola stampare, vediamolo con un esempio:

```
#definiamo una variabile
c = 3.141592653589793

#stampa come intero
print('%d' %c)

#stampa come reale
print('%f' %c) #di default stampa solo prime 6 cifre
print(f'{c}')  #di default stampa tutte le cifre

#per scegliere il numero di cifre, ad esempio sette cifre
print('%.7f' %c)
print(f'{c:.7f}')

[Output]
3
3.141593
3.141592653589793
3.1415927
3.1415927
```

Notare che il computer esegue un arrotondamento. Inoltre abbiamo usato la lettera "f" perché vogliamo un numero decimale, se volessimo altri formati potremmo usare: "i" o "d" per gli interi, "o" per un numero in base otto, "x" per un numero esadecimale, "e" per un numero in notazione scientifica. Infine le lettere "c" e "s" indicano rispettivamente un singolo carattere e una stringa.
Una variabile può essere ridefinita e cambiare valore, addirittura cambiate tipo, il computer userà l'assegnazione più recente (attenzione mi raccomando che ci vuole poco che una cosa del genere fornisca errori):

```
#definiamo una variabile
x = 30

"""

operazioni varie


"""

#ridefiniamo la variabile
```

```
x = 18

print('x=', x)

"""
E' possibile anche sovrascrivere una variabile
con un numero che dipende dal suo valore precedente:
"""
x = x + 1 #incrementiamo di uno
#Oppure:
x += 1
print('x=', x)

x = x * 2 #moltiplichiamo per due
#Oppure:
x *= 2
print('x=', x)

[Output]
x= 18
x= 20
x= 80
```

Come si vede i vari comandi $x = x$ *operazione numero* possono essere abbreviati con x *operazione*$=$ *numero*.

5.4 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 (ogni tanto però è interessante andare a vedere cosa ci sta dietro, ma ne parleremo, non molto, ma in vari ambiti, più avanti). 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:

```
import numpy

#per usare un contenuto di questa libreria basta scrivere numpy.contenuto
pigreco = numpy.pi
print(pigreco)

#Possiamo anche ribattezzare le librerie in questo modo
import numpy as np
#da ora all'interno del codice numpy si chiama np

eulero = np.e
print(eulero)

[Output]
3.141592653589793
2.718281828459045
```

```
import math

coseno=math.cos(0)
seno = math.sin(np.pi/2) #python usa di default gli angoli in radianti!!!
senosbagliato = math.sin(90)

print('Coseno di 0=', coseno, "\nSeno di pi/2=", seno, "\nSeno di 90=", senosbagliato)

#bisogna quindi stare attenti ad avere tutti gli angoli in radianti
angoloingradi = 45
#questa funzione converte gli angoli da gradi a radianti
angoloinradianti = math.radians(angoloingradi)

print("Angolo in gradi:", angoloingradi, "Angolo in radianti:", angoloinradianti)

[Output]
Coseno di 0= 1.0
Seno di pi/2= 1.0
Seno di 90= 0.8939966636005579
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). Altra cosa interessante è che essendo la documentazione scritta sul codice, esattamente come qui noi scriviamo i commenti, potete consultarla anche da shell. Su Pyzo vi basta scrive sulla shell: "nomefunzione?", mentre se usate una shell normale, stile quella di ubuntu dovete prima digitare "python" oppure "python3" sulla shell e vi si aprirà l'ambiente python dove potete importare i pacchetti come se scriveste codice e per vedere la documentazione vi basta fare "help(nomefunzione)". Inoltre spesso si trova la sintassi:

```
form numpy import *
```

con numpy o con qualsiasi altra libreria. Questo ci permette di non dover scrivere ogni volta "numpy." o "np." davanti le funzioni che vogliamo utilizzare. Bisogna però stare attenti all'esistenza di funzioni con stesso nome ma che fanno cose diverse, come dicevamo prima la funzione seno ad esempio. In un contentesto in cui si importano sia math che numpy usando l'asterisco ci sarà un conflitto su che funzione usare; o meglio, verrà usata la funzione della libreria più recentemente importata. il che vuol dire che se scrivete:

```
form numpy import *
from math import *
```

la funzione seno chiamata sarà quella di math e quindi avret un errore se il possibile input non divessere essere un numero ma un array.

5.5 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 finita 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.

```
Traceback (most recent call last):
  File "<tmp 1>", line 5, in <module>
    b = 1/a
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.
Ora capisco che prima vi dico di leggerlo al contrario e poi qui ve lo descrivo nel normale ordine di lettura, ma è solo per farvi capire il significato delle varie linee di errore. Facciamo adesso un esempio un po' più complicato, leggendolo correttamente, in cui useremo funzioni quindi magari potete tornare a rileggerlo dopo se volete.

```
def err(c):
    b = 1/c
    return b

def run(c):
    print(err(c))

if __name__ == "__main__":
    run(0)

[Output]
Traceback (most recent call last):
  File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 9, in
    <module>
    run(0)
  File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 6, in
    run
    print(err(c))
  File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 2, in
    err
    b = 1/c
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. Se proprio non capite che vi sta dicendo, buona norma è copiare la riga più bassa del traceback e incollarla su Google. Qualcuno avrà già avuto il vostro problema.

5.6 Un paio di trick

Abbiamo visto che Python non necessita di punti e virgola per delimitare una linea come in C. Tuttavia è possibile usarli con lo stesso scopo, quindi se volete mettere due comandi Python sulla stessa riga vi basta separarli con un ";" e verrano eseguiti come fossero due righe diverse. Inoltre possono essere usate sulla shell per nascondere l'output del comando, dato che come premete invio sulla shell la riga appena scritta viene interpretata, esattamente come su Mathematica. Sempre su shell invece se avete bisogno di fare dei calcoli veloci stile calcolatrice potete usare l'underscore per riferirvi al risultato precedente, come quando sulla calcolatrice premete il tasto Ans.