Introduzione a Python

Print to PDF

Contents

Python è un linguaggio di programmazione ad alto livello, interpretato e pensato per la programmazione “general purpose”. Python supporta un sistema dei tipi

dimanico e diversi paradigmi di programmazione tra cui la programmazione orientata agli oggetti, la programmazione imperativa, la programmazione funzionale e la programmazione procedurale. Il linguaggio è stato ideato nel 1991 da Guido van Rossum e il suo nome è ispirato alla serie TV satirica Monty Python's Flying Circus (https://en.wikipedia.org/wiki/Monty_Python's_Flying_Circus).

Benché Python non sia nato come linguaggio di programmazione per il calcolo scientifico, la sua estrema versatilità ha contribuito al nascere di una serie di librerire che rendono la computazione numerica in Python comoda ed efficiente. Buona parte di queste librerie fanno parte di "SciPy" (https://www.scipy.org/), un ecosistema di software open-source per il calcolo scientifico. In questo laboratorio, oltre a una introduzione a Python, vedremo in particolare i fondamenti di NumPy (calcolo scientifico) e Matplolib (Plot 2D).

Referenze importanti da consultare durante il corso, solo le seguenti documentazioni:

- Python 3: https://docs.python.org/3/index.html;
- Numpy: http://www.numpy.org/;
- Matplotlib: https://matplotlib.org/.

2.1. Numeri

I tipi di dato numerici in Python sono int, float e complex. Noi ci concentreremo su int e float. Alcuni esempi di operazioni tra numeri:

```
3 + 5 #somma
```

```
8
```

```
2 - 8 #differenza
```

```
-6
```

```
3 * 5 #prodotto
```

```
15
```

```
3 / 2 #divisione, da notare che in Python 3, la divisione tra numeri
```

```
1.5
```

```
3 // 2 #divisione intera
```

```
1
```

```
9 ** 2 #elevamento a potenza
```

```
81
```

```
4 % 2 #modulo
```

```
0
```

```
(1 + 4) * (3 - 2) #uso delle parentesi
```

```
5
```

2.2. Variabili e Tipi

In generale, i numeri senza virgola vengono interpretati come int, mentre quelli con virgola come float. Dato che Python è tipizzato dinamicamente, non dobbiamo esplicitamente dichiarare il tipo di una variabile. Il tipo verrà associato alla variabile non appena vi assegniamo un valore. Possiamo controllare il tipo di una variabile mediante la funzione `type`:

```
x = 3
y = 9
z = 1.5
h = x/y
l = x//y
type(x), type(y), type(z), type(h), type(l)
```

```
(int, int, float, float, int)
```

E' possibile effettuare il casting da un tipo a un altro mediante le funzioni `int` e `float`:

```
int(2.5)
```

```
2
```

```
float(3)
```

```
3.0
```

Come in C, sono definite le operazioni "in place" tra variabili:

```
x = 8
y = 12
x+=y #del tutto equivalente a x=x+y
x
```

```
20
```

Non sono definite le notazioni “++” e “−”.

```
a++ #error!
```

Per effettuare un incremento di una unità, va utilizzata la notazione +=1:

```
a=1
a+=1
a
```

```
2
```

> **Domanda 1**
>
> Qual è il tipo della seguente variabile?
>
> ```
> x = (3//2*2)**2+(3*0.0)
> ```

2.3. Booleani

I booleani vengono rappresentati mediante le parole chiave `True` e `False` (iniziano entrambe per maiuscola).

```
print(True)
print(False)
```

```
True
False
```

```
type(True)
```

```
bool
```

E’ possibile generare booleani mediante gli operatori di confronto:

```
5==5
```

```
True
```

```
7==5
```

```
False
```

```
5>4
```

```
True
```

```
9>10
```

```
False
```

```
9<=10
```

```
True
```

```
11<=10
```

```
False
```

Gli operatori logici sono `and` e `or`:

```
print(5==5 and 3<5)
print(3>5 or 3<5)
print(3>9 or 3<2)
```

```
True
True
False
```

E' possibile effettuare in controllo sui tipi mediante `type` e `==`:

```
type(2.5)==float
```

```
True
```

```
type(2)==float
```

```
False
```

In alternativa, è possibile utilizzare la funzione `isinstance`:

```
isinstance(2.5,float)
```

```
True
```

```
isinstance(2.5,int)
```

```
False
```

La funzione `isinstance` è particolarmente comoda quando si vuole controllare che una variabile appartenga a uno tra una serie di tipi. Ad esempio, se vogliamo controllare che una variabile contenga un numero:

```
isinstance(2.5,(float,int))
```

```
True
```

```
isinstance(5,(float,int))
```

```
True
```

2.4. Stampa

La stampa avviene mediante la funzione `print`:

```
var = 2.2
print(var)
```

```
2.2
```

Possiamo stampare una riga vuota omettendo il parametro di print:

```
print(2)
print()
print(3)
```

```
2

3
```

Alternativamente, possiamo specificare di inserire due "a capo" alla fine della stampa specificando il parametro `end="\n\n"` ("\n\n" è una stringa - approfondiremo le stringhe in seguito):

```
print(2, end="\n\n")
print(3)
```

```
2

3
```

Lo stesso metodo può essere usato per omettere l'inserimento di spazi tra due stampe consecutive:

```
print(2, end="") #"" rappresenta una stringa vuota
print(3)
```

```
23
```

Possiamo stampare più elementi di seguito separando gli argomenti di `print` con delle virgole. Inoltre, la funzione `print` permette di stampare anche numeri, oltre a stringhe:

```
print(1,8/2,7,14%6,True)
```

```
1 4.0 7 2 True
```

2.5. Liste

Le liste sono una struttura dati di tipo sequenziali che possono essere utilizzate per rappresentare sequenze di valori di qualsiasi tipo. Le liste possono anche contenere elementi di tipi misti. Una lista si definisce utilizzando le parentesi quadre:

```
l = [1,2,3,4,5] #questa è una lista (parentesi quadre)
print(l)
```

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

Le liste possono essere indicizzate utilizzando le parentesi quadre. L'indicizzazione inizia da 0 come in C:

```
print(l[0],l[2])
l[0]=8 #assegnamento di un nuovo valore alla prima locazione di memor
print(l)
```

```
1 3
[8, 2, 3, 4, 5]
```

E' possibile aggiungere nuovi valori a una lista mediante la funzione `append`:

```
l = []
print(l)
l.append(1)
l.append(2.5)
l.append(8)
l.append(-12)
print(l)
```

```
[]
[1, 2.5, 8, -12]
```

Le liste possono essere concatenate mediante l'operatore somma:

```
l1 = [1,5]
l2 = [4,6]
print(l1+l2)
```

```
[1, 5, 4, 6]
```

L'operatore di moltiplicazione può essere utilizzato per ripetere una lista. Ad esempio:

```
l1 = [1,3]
print(l1*2) #concatena l1 a se stessa per due volte
```

```
[1, 3, 1, 3]
```

Utilizzando l'operatore di moltiplicazione, è possibile creare velocemente liste con un numero arbitrario di valori uguali. Ad esempio:

```
print([0]*5) #lista di 5 zeri
print([0]*4+[1]*1) #4 zeri seguiti da 1 uno
```

```
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 1]
```

La lunghezza di una lista può essere ottenuta utilizzando la funzione `len`:

```
print(l2)
print(len(l2))
```

```
[4, 6]
2
```

Sulle liste è definito un ordinamento che dipende dalla loro lunghezza: liste più corte sono "minori di" liste più lunghe:

```
print([1,2,3]<[1,2,3,4])
print([1,2,3,5]>=[1,2,3,4])
```

```
True
True
```

L'operatore `==` non controlla se le lunghezze sono uguali, ma verifica che il contenuto delle due liste sia effettivamente uguale:

```
print([1,2,3]==[1,2,3])
print([1,2,3]==[1,3,2])
```

```
True
False
```

E' possibile controllare che un elemento appartenga alla lista mediante la parola chiave `in`:

```
print(7 in [1,3,4])
print(3 in [1,3,4])
```

```
False
True
```

Le funzioni `max` e `min` possono essere utilizzate per calcolare il massimo e il minimo di una lista:

```
l=[-5,2,10,6]
print(max(l))
print(min(l))
```

```
10
-5
```

E' possibile rimuovere un valore da una lista mediante il metodo `remove`:

```
l=[1,2,3,4,2]
print(l)
l.remove(2)
print(l)
```

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

Tale metodo tuttavia rimuove solo **la prima occorrenza** del valore passato. Se vogliamo rimuovere un valore identificato da uno specifico indice, possiamo usare il costrutto `del`:

```
l=[1,2,3,4,2]
print(l)
del l[4]
print(l)
```

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

Inoltre, per accedere all'ultimo elemento e rimuoverlo, possiamo usare il metodo `pop`:

```
l=[1,2,3,4,5]
print(l)
print(l.pop())
print(l)
```

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

2.5.1. Indicizzazione e Slicing

E' possibile estrarre una sottolista da una lista specificando il primo indice (incluso) e l'ultimo indice (escluso) separati dal simbolo `:`. Questa notazione è in qualche modo reminiscende del metodo `substr` delle stringhe di C++.

```
l = [1,2,3,4,5,6,7,8]
print("Lista l     ->", l)
print("l[0:3]      ->", l[0:3]) #dall'indice 0 (incluso) all'indice 3
print("l[1:2]      ->", l[1:2]) #dall'indice 1 (incluso) all'indice 2
```

```
Lista l     -> [1, 2, 3, 4, 5, 6, 7, 8]
l[0:3]      -> [1, 2, 3]
l[1:2]      -> [2]
```

Quando il primo indice è omesso, questo viene automaticamente sostituito con "0":

```
print("l[:2]       ->", l[:2]) #dall'indice 0 (incluso) all'indice 2
#equivalente al seguente:
print("l[0:2]      ->", l[0:2]) #dall'indice 0 (incluso) all'indice 2
```

```
l[:2]       -> [1, 2]
l[0:2]      -> [1, 2]
```

Analogamente, se omettiamo il secondo indice, esso viene sostituito con l'ultimo indice della lista:

```
print("Ultimo indice della lista:",len(l))
print("l[3:]       ->", l[3:]) #dall'indice 3 (incluso) all'indice 5
#equivalente al seguente:
print("l[3:5]      ->", l[3:8]) #dall'indice 3 (incluso) all'indice 5
```

```
Ultimo indice della lista: 8
l[3:]       -> [4, 5, 6, 7, 8]
l[3:5]      -> [4, 5, 6, 7, 8]
```

Omettendo entrambi gli indici:

```
print("l[:]         ->", l[:]) #dall'indice 0 (incluso) all'indice 5
#equivalente a:
print("l[0:8]       ->", l[0:8]) #dall'indice 0 (incluso) all'indice
```

```
l[:]         -> [1, 2, 3, 4, 5, 6, 7, 8]
l[0:8]       -> [1, 2, 3, 4, 5, 6, 7, 8]
```

E' inoltre possibile specificare il "passo", come terzo numero separato da un altro simbolo `:`:

```
print("l[0:8:2]    ->", l[0:8:2])
#da 0 (incluso) a 8 (escluso) a un passo di 2 (un elemento sì e uno n
#equivalente a:
print("l[::2]      ->", l[0:8:2])
#da 0 (incluso) a 8 (escluso) a un passo di 2 (un elemento sì e uno n
```

```
l[0:8:2]    -> [1, 3, 5, 7]
l[::2]      -> [1, 3, 5, 7]
```

Per invertire l'ordine degli elementi, è inoltre possibile specificare un passo negativo. In questo caso, bisogna assicurarsi che il primo indice sia maggiore del secondo:

```
print("l[5:2:-1]    ->", l[5:2:-1])
#da 5 (incluso) a 2 (escluso) a un passo di -1
print("l[2:5:-1]    ->", l[2:5:-1])
#in questo caso, il primo indice è più piccolo del secondo,
#quindi il risultato sarà una lista vuota
```

```
l[5:2:-1]    -> [6, 5, 4]
l[2:5:-1]    -> []
```

Anche in questo caso, omettendo degli indici, questi verranno rimpiazzati con le scelte più ovvie. Nel caso dell'omissione però, cambiano le condizioni di inclusione ed esclusione degli indici. Vediamo qualche esempio:

```
print("l[:2:-1]     ->", l[:2:-1])
#dall'ultimo indice (incluso) a 2 (escluso) a un passo di -1
#equivalente a:
print("l[8:2:-1]    ->", l[8:2:-1])

print()
print("l[3::-1]     ->", l[3::-1])
#dal terzo indice (incluso) a 0 (incluso, in quanto omesso) a un pass
#simile, ma non equivalente a:
print("l[3:0:-1]    ->", l[3:0:-1])
#dal terzo indice (incluso) a 0 (escluso) a un passo di -1

print()
print("l[::-1]      ->", l[::-1])
#dall'ultimo indice (incluso) al primo (incluso, in quanto omesso) a
#simile, ma non equivalente a:
print("l[8:0:-1]    ->", l[8:0:-1])
#dall'ultimo indice (incluso) al primo (escluso) a un passo di -1
```

```
l[:2:-1]     -> [8, 7, 6, 5, 4]
l[8:2:-1]    -> [8, 7, 6, 5, 4]

l[3::-1]     -> [4, 3, 2, 1]
l[3:0:-1]    -> [4, 3, 2]

l[::-1]      -> [8, 7, 6, 5, 4, 3, 2, 1]
l[8:0:-1]    -> [8, 7, 6, 5, 4, 3, 2]
```

La notazione `::-1`, in particolare, è utile per invertire le liste:

```
print(l)
print(l[::-1])
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
[8, 7, 6, 5, 4, 3, 2, 1]
```

Indicizzazione e slicing possono essere utilizzate anche per assegnare valori agli elementi delle liste. Ad esempio:

```
l = [5,7,9,-1,2,6,5,4,-6]
print(l)
l[3]=80
print(l)
```

```
[5, 7, 9, -1, 2, 6, 5, 4, -6]
[5, 7, 9, 80, 2, 6, 5, 4, -6]
```

E' anche possibile assegnare più di un elemento alla volta:

```
l[::2]=[0,0,0,0,0] #assegno 0 ai numeri di posizione dispari
print(l)
```

```
[0, 7, 0, 80, 0, 6, 0, 4, 0]
```

Le liste possono anche essere annidate:

```
a1 = [1,2,[4,8,[7,5]],[9],2]
print(a1)
```

```
[1, 2, [4, 8, [7, 5]], [9], 2]
```

L'indicizzazione di queste strutture annidate avviene concatenando gli indici come segue:

```
print(a1[2][2][0]) #il primo indice seleziona la lista [4,8,...]
#il secondo indice seleziona la lista [7,5]
#il terzo indice seleziona l'elemento 7
```

```
7
```

> 🙋‍♂️ **Domanda 2**

> Estrarre la lista `[3, 1.2]` dalla seguente lista:
>
> ```
> l = [1, 4, 5, [7, 9, -1, [0, 3, 2, 1.2], 8, []]]
> ```

2.6. Tuple

Le tuple sono simili alle liste, ma sono immutabili. Non possono cioè essere modificate dopo la loro inizializzazione. A differenza delle liste, le tuple vengono definite utilizzando le parentesi tonde:

```
l = [1,2,3,4,5] #questa è una lista (parentesi quadre)
t = (1,2,3,4,5) #questa è una tupla (parentesi tonde)
print(l)
print(t)
```

```
[1, 2, 3, 4, 5]
(1, 2, 3, 4, 5)
```

Le regole di indicizzazione e slicing viste per le liste valgono anche per tuple. Tuttavia, come accennato prima, le tuple non possono essere modificate:

```
t = (1,3,5)
t[0]=8 #restituisce un errore in quanto le tuple sono immutabili
```

Inizializzare una tupla con un solo elemento produrrà un numero. Ciò avviene in quanto le parentesi tonde vengono utilizzate anche per raggruppare i diversi termini di una operazione:

```
t=(1)
print(t)
```

```
1
```

Per definire una tupla monodimensionale, dobbiamo aggiungere esplicitamente una virgola, dopo il primo elemento:

```
t=(1,)
print(t)
```

```
(1,)
```

E' inoltre possibile omettere le parentesi nella definizione delle tuple:

```
t1=1,3,5
t2=1,
print(t1,t2)
```

```
(1, 3, 5) (1,)
```

E' possibile convertire tuple in liste e viceversa:

```
l=[1,2,3,4,5,6,7,8]
t=(4,5,6,7,4,8,2,4)
ttl = list(t)
ltt = tuple(l)

print(ttl)
print(ltt)
```

```
[4, 5, 6, 7, 4, 8, 2, 4]
(1, 2, 3, 4, 5, 6, 7, 8)
```

Le tuple possono inoltre essere create e "spacchettate" al volo:

```
t1=1,2,3

print(t1)

a,b,c=t1 #spacchettamento della tupla
print(a,b,c)
```

```
(1, 2, 3)
1 2 3
```

Questo sistema permette di effettuare lo swap di due variabili in una sola riga di codice:

```
var1 = "Var 1"
var2 = "Var 2"

print(var1,var2)

var1,var2=var2,var1
print(var1,var2)

#equivalente a:
var1 = "Var 1"
var2 = "Var 2"
t = (var2,var1)
var1=t[0]
var2=t[1]
print(var1,var2)
```

```
Var 1 Var 2
Var 2 Var 1
Var 2 Var 1
```

Le tuple annidate possono essere spacchetate come segue:

```
t = (1,(2,3),(4,5,6))
x,(t11,t12),(t21,t22,t23) = t
print(t)
print(x, t11, t12, t21, t22, t23)
#La notazione a,b,c,d,e,f = t restituirebbe un errore
```

```
(1, (2, 3), (4, 5, 6))
1 2 3 4 5 6
```

> 🤦 **Domanda 3**
>
> Qual è la differenza principale tra tuple e liste? Si faccia l'esempio di un caso

> in cui una tupla è un tipo più appropriato rispetto a una lista.

2.7. Stringhe

In Python possiamo definire le stringhe in tre modi:

```
s1 = 'Singoli apici'
s2 = "Doppi apici, possono contenere anche apici singoli '' "
s3 = """Tripli
doppi apici
possono essere definite su più righe"""
type(s1), type(s2), type(s3)
```

```
(str, str, str)
```

La stampa avviene mediante la funzione “print”:

```
print(s1)
print(s2)
print(s3)
```

```
Singoli apici
Doppi apici, possono contenere anche apici singoli ''
Tripli
doppi apici
possono essere definite su più righe
```

Le stringhe hanno inoltre una serie di metodi predefiniti:

```
print("ciao".upper()) #rendi tutto maiuscolo
print("CIAO".lower()) #tutto minuscolo
print("ciao come stai".capitalize()) #prima lettera maiuscola
print("ciao come stai".split()) #spezza una stringa e restituisce una
print("ciao, come stai".split(','))# spezza quando trova la virgola
print("-".join(["uno","due","tre"])) #costruisce una stringa concaten
#della lista e separandoli mediante il delimitatore
```

```
CIAO
ciao
Ciao come stai
['ciao', 'come', 'stai']
['ciao', ' come stai']
uno-due-tre
```

Le stringhe possono essere indicizzate in maniera simile agli array per ottenere delle sottostringhe:

```
s = "Hello World"
print(s[:4]) #primi 4 caratteri
print(s[4:]) #dal quarto carattere alla fine
print(s[4:7]) #dal quarto al sesto carattere
print(s[::-1]) #inversione della stringa
```

```
Hell
o World
o W
dlroW olleH
```

Il metodo `split` in particolare può essere utilizzato per la tokenizzazione o per estrarre sottostringhe in maniera agevole. Ad esempio, supponiamo di voler estrarre il numero `2018` dalla stringa `A-2017-B2`:

```
print("A-2017-B2".split('-')[1])
```

```
2017
```

L'operatore `==` controlla che due stringhe siano uguali:

```
print("ciao"=="ciao")
print("ciao"=="ciao2")
```

```
True
False
```

Gli altri operatori rispecchiano l'ordinamento lessicografico tra stringhe:

```
print("abc"<"def")
print("Abc">"def")
```

```
True
False
```

> 🙋 **Domanda 4**
>
> Quale codice permette di manipolare la stringa `azyp-kk9-382` per ottenere la stringa `Kk9`?

2.8. Formattazione di Stringhe

Possiamo costruire stringhe formattate seguendo una sintassi simile a quella di printf:

```
#per costruire la stringa formattata, faccio seguire la stringa dal s
#una tupla contenente gli argomenti
s1 = "Questa %s è formattata. Posso inserire numeri, as esempio %0.2f
print(s1)
```

```
Questa stringa è formattata. Posso inserire numeri, as esempio 3.00
```

Un modo alternativo e più recente di formattare le strighe consiste nell'usare il metodo "format":

```
s2 = "Questa {} è formattata. Posso inserire numeri, ad esempio {}"\
    .format("stringa",3.000002)# il carattere "\" permette di spezzar
print(s2)
```

```
Questa stringa è formattata. Posso inserire numeri, ad esempio 3.0000(
```

E' possibile specificare il tipo di ogni argomento utilizzando i due punti:

```
print("Questa {:s} è formattata. Posso inserire numeri, ad esempio {:
      .format("stringa",3.00002)) #parametri posizionali, senza speci
```

```
Questa stringa è formattata. Posso inserire numeri, ad esempio 3.00
```

E' anche possibile assegnare nomi agli argomenti in modo da richimarli in maniera non ordinata:

```
print("Questa {str:s} è formattata. Posso inserire numeri, ad esempio
    .format(num=3.00002, str="stringa"))
```

```
Questa stringa è formattata. Posso inserire numeri, ad esempio 3.00
```

> 🙋‍♂️ **Domanda 5**
>
> Date le variabili:
>
> ```
> a = "hello"
> b = "world"
> c = 2.0```
>
> Usare la formattazione delle stringhe per stampare la stringa `he
>
> ```

2.9. Dizionari

I dizionari sono simili a delle liste, ma vengono indicizzate da oggetti di tipo "hashable", ad esempio stringhe:

```
d = {"val1":1, "val2":2}
print(d)

print(d["val1"])
```

```
{'val1': 1, 'val2': 2}
1
```

E' possibile ottenere la lista delle chiavi e dei valori come segue:

```
print(d.keys()) #chiavi e valori sono in ordine casuale
print(d.values())
```

```
dict_keys(['val1', 'val2'])
dict_values([1, 2])
```

E' possibile indicizzare dizionari con tuple (che sono "hashable")

```
d = {(2,3):5, (4,6):11}

print(d[(2,3)])
```

```
5
```

I dizionari possono anche essere estesi dinamicamente:

```
d = dict() #dizionario vuoto
d["chiave"]="valore"
print(d)
```

```
{'chiave': 'valore'}
```

Possiamo controllare che un elemento si trovi tra le chiavi di un dizionario come segue:

```
d = {1:'ciao', '5': 5, 8: -1}
print(5 in d)
print('5' in d)
```

```
False
True
```

E' possibile controllare che un elemento si trovi tra i valori di un dizionario come segue:

```
print(-1 in d.values())
```

```
True
```

2.10. Set

I set sono delle strutture dati che possono contenere solo una istanza di un dato elemento:

```
s = {1,2,3,3}
print(s) #può essere contenuto solo un "3"
```

```
{1, 2, 3}
```

Possiamo aggiungere un elemento a un set mediante il metodo "add":

```
print(s)
s.add(5)
print(s)
s.add(1) #non ha effetto. 1 è già presente
print(s)
```

```
{1, 2, 3}
{1, 2, 3, 5}
{1, 2, 3, 5}
```

Anche in questo caso, possiamo controllare l'appartenenza di un elemento ad un set mediante la parola chiave `in`:

```
s={1,5,-1}
print(-1 in s)
print(8 in s)
```

```
True
False
```

E' inoltre possibile creare set da liste:

```
set([1,3,3,2,5,1])
```

```
{1, 2, 3, 5}
```

2.11. Costrutti if/elif/else

I costrutti condizionali funzionano in maniera simile ai linguaggi basati su C. A differenza di tali linguaggi tuttavia, Python sostituisce le parentesi con l'indentazione obbligatoria. Il seguente codice C++:

```
int var1 = 5;
int var2 = 10;
if(var1<var2) {
    int var3 = var1+var2;
    cout << "Hello World "<<var3;
}
cout << "End";```
viene invece scritto come segue:
```

```
var1 = 5
var2 = 10
if var1<var2:
    var3 = var1+var2
    print("Hello World",var3)
print("End")
```

```
Hello World 15
End
```

In pratica:

- la condizione da verificare non è racchiusa tra parentesi;
- i due punti indicano l'inizio del corpo dell'if;
- l'indentazione stabilisce cosa appartiene al corpo dell'if e cosa non appartiene al corpo dell'if.

Dal momento che l'indentazione ha valore sintattico, essa diventa obbligatoria. Inoltre, non è possibile indentare parti di codice ove ciò non è significativo. Ad esempio, il seguente codice restituisce un errore:

```
print("Hello")
    print("World") #errore di indentazione
```

Le regole di indentazione appena viste, valgono anche per i cicli e altri costrutti in cui è necessario deliminare blocchi di codice. Il costrutto if, permette anche di specificare un ramo `else` e un ramo `elif` per i controlli in cascata. Vediamo alcuni esempi:

```
true_condition = True
false_condition = False

if true_condition: #i due punti ":" sono obbligatori
    word="cool!"
    print(word) #l'indentazione è obbligatoria

if false_condition:
    print("not cool :(")

if not false_condition: #neghiamo la condizione con "not"
    word="cool"
    print(word,"again :)")

if false_condition:
    word="this"
    print(word+" is "+"false")
else: #due punti + indentazione
    print("true")

if false_condition:
    print("false")
elif 5>4: #implementa un "else if"
    print("5>4")
else:
    print("4<5??")
```

```
cool!
cool again :)
true
5>4
```

E’ anche possibile verificare se un valore appartiene a una lista. Ciò è molto utile per verificare se un parametro è ammesso.

```
allowed_parameters = ["single", "double", 5, -2]

parameter = "double"

if parameter in allowed_parameters:
    print(parameter,"is ok")
else:
    print(parameter,"not in",allowed_parameters)

x=8
if x in allowed_parameters:
    print(x,"is not ok")
else:
    print(x,"not in",allowed_parameters)
```

```
double is ok
8 not in ['single', 'double', 5, -2]
```

Una variante "inline" del costrutto if può essere utilizzata per gli assegnamenti condizionali:

```
var1 = 5
var2 = 3
m = var1 if var1>var2 else var2 #calcola il massimo
print(m)
```

```
5
```

In Python non esiste il costrutto switch, per implementare il quale si utilizza una lista di elif in cascata:

```
s = "ciao"
if s=="help":
    print("help")
elif s=="world":
    print("hello",s)
elif s=="ciao":
    print(s,"mondo")
else:
    print("Default")
```

```
ciao mondo
```

> 🙋 **Domanda 6**
>
> Si consideri il seguente codice:
>
> ```
> x=2
> if x>0:
>     y=12
>     x=x+y
>     print y
> else:
>         z=28
>         h=12
>         print z+h
> ```
>
> Si evidenziino eventuali errori sintattici. Si riscriva il codice in C++ o Java e si confrontino le due versioni del codice.

2.12. Cicli while e for

I cicli while, si definiscono come segue:

```
i=0
while i<5: #due punti ":"
    print(i) #indentazione
    i+=1
```

```
0
1
2
3
4
```

La sintassi dei cicli for è un po' diversa dalla sintassi standard del C. I cicli for in Python sono più simili a dei **foreach** e richiedono un "iterable" (ad esempio una lista o una tupla) per essere eseguiti:

```
l=[1,7,2,5]
for v in l:
    print(v)
```

```
1
7
2
5
```

Per scrivere qualcosa di equivalente al seguente codice C:

```
for (int i=0; i<5; i++) {...}
```

possiamo utilizzare la funzione **range** che genera numeri sequenziali al volo:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Range non genera direttamente una lista, ma un "generator" di tipo range, ovvero un oggetto capace di generare numeri. Se vogliamo convertirlo in una lista, dobbiamo farlo esplicitamente:

```
print(range(5)) #oggetto di tipo range, genera numeri da 0 a 5 (esclu
print(list(range(5))) #range non fa altro che generare numeri consecu
#convertendo range in una lista, possiamo verificare quali numeri ven
```

```
range(0, 5)
[0, 1, 2, 3, 4]
```

Se volessimo scorrere contemporaneamente indici e valori di un array potremmo

scrivere:

```
array=[1,7,2,4,5]
for i in range(len(array)):
    print(i,"->",array[i])
```

```
0 -> 1
1 -> 7
2 -> 2
3 -> 4
4 -> 5
```

In Python però, è possibile utilizzare la funzione `enumerate` per ottenere lo stesso risultato in maniera più compatta:

```
for index,value in enumerate(array): #get both index and value
    print(index,"->",value)
```

```
0 -> 1
1 -> 7
2 -> 2
3 -> 4
4 -> 5
```

Supponiamo adesso di voler scorrere contemporaneamente tutti gli iesimi elementi di più liste. Ad esempio:

```
a1=[1,6,2,5]
a2=[1,8,2,7]
a3=[9,2,5,2]

for i in range(len(a1)):
    print(a1[i],a2[i],a3[i])
```

```
1 1 9
6 8 2
2 2 5
5 7 2
```

Una funzione molto utile quando si lavora con i cicli è `zip`, che permette di raggruppare gli elementi corrispondenti di diverse liste:

```
l1 = [1,6,5,2]
l2 = [3,8,9,2]
zipped=list(zip(l1,l2))
print(zipped)
```

```
[(1, 3), (6, 8), (5, 9), (2, 2)]
```

In pratica, `zip` raggruppa gli elementi i-esimi delle liste in tuple. La i-esima tupla di `zipped` contiene gli i-esimi elementi delle due liste. Combinando `zip` con un ciclo for, possiamo ottenere il seguente risultato:

```
for v1,v2,v3 in zip(a1,a2,a3):
    print(v1,v2,v3)
```

```
1 1 9
6 8 2
2 2 5
5 7 2
```

che è equivalente al codice visto in precedenza. E' anche possibile combinare zip e enumerate come segue:

```
for i,(v1,v2,v3) in enumerate(zip(a1,a2,a3)):
    print(i,"->",v1,v2,v3)
```

```
0 -> 1 1 9
1 -> 6 8 2
2 -> 2 2 5
3 -> 5 7 2
```

Attenzione, anche `zip`, come `range`, produce un generator. Pertanto è necessario convertirlo esplicitamente in una lista per stamparne i valori:

```
print(zip(l1,l2))
print(list(zip(l1,l2)))
```

```
<zip object at 0x103f7c740>
[(1, 3), (6, 8), (5, 9), (2, 2)]
```

2.13. Comprensione di liste e dizionari

La comprensione di liste è uno strumento sintattico che permette di definire liste al volo a partire da altre liste, in maniera iterativa. Ad esempio, è possibile moltiplicare tutti gli elementi di una lista per un valore, come segue:

```
a = list(range(8))

b = [x*3 for x in a] #la lista "a" viene iterata. La variabile "x" co

print(a)
print(b)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 3, 6, 9, 12, 15, 18, 21]
```

E' anche possibile includere solo alcuni elementi selettivamente:

```
print([x for x in a if x%2==0]) #include solo i numeri pari
```

```
[0, 2, 4, 6]
```

```
a = [1,3,8,2,9]
b = [4,9,2,1,4]

c = [x+y for x,y in zip(a,b)]

print(a)
print(b)
print(c) #i suoi elementi sono le somme degli elementi di a e b
```

```
[1, 3, 8, 2, 9]
[4, 9, 2, 1, 4]
[5, 12, 10, 3, 13]
```

I meccanismi di comprensione si possono utilizzare anche nel caso dei dizionari:

```
a = ["one","two","three","four","five","six","seven"]
b = range(1,8)
d = {i:s for i,s in zip(a,b)}
print(d)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6, 'sev
```

> 🙋 **Domanda 7**
>
> Tutte le operazioni che si possono fare mediante comprensione di liste e dizionari possono essere fatte mediante un ciclo for? Quali sono i vantaggi principali di queste tecniche rispetto a l'utilizzo dei cicli for?

2.14. Definizione di Funzioni

Considerato quanto già detto sulla indentazione, la definizione di una funzione è molto naturale:

```
def fun(x, y):
    return x**y

print(fun(3,2)) #il valore di default "2" viene utilizzato
```

```
9
```

Inoltre, in maniera simile a quanto avviene con il linguaggio C, è possibile definire valori di default per i parametri:

```
def fun(x, y=2):
    return x**y

print(fun(3)) #il valore di default "2" viene utilizzato
```

```
9
```

I parametri di una funzione possono essere specificati in un ordine diverso rispetto a quello in cui essi sono stati definiti richiamandone il nome:

```
print(fun(y=3,x=2))
```

```
8
```

E' possibile definire una funzione che restituisce più di un elemento utilizzando le tuple:

```
def soMuchFun(x,y):
    return x**y, y**x

print(soMuchFun(2,3))

a,b=soMuchFun(2,3) #posso "spacchettare" la tupla restituita
print(a,b)
```

```
(8, 9)
8 9
```

E' inoltre possibile definire funzioni anonime come segue:

```
myfun = lambda x: x**2 #un input e un output
print(myfun(2))

myfun1 = lambda x,y: x+y #due input e un output
print(myfun1(2,3))

myfun2 = lambda x,y: (x**2,y**2) #due input e due output
print(myfun2(2,3))
```

```
4
5
(4, 9)
```

> 🙋 **Domanda 8**
>
> Qual è il vantaggio di definire una funzione mediante **lambda**? Quali sono i suoi limiti?

2.15. Map e Filter

Le funzioni **map** e **filter** permettono di eseguire operazioni sugli elementi di una lista. In particolare, **map** applica una funzione a tutti gli elementi di una lista:

```
def pow2(x):
    return(x**2)

l1 = list(range(6))
l2 = list(map(pow2,l1)) #applica pow2 a tutti gli elementi della list
print(l1)
print(l2)
```

```
[0, 1, 2, 3, 4, 5]
[0, 1, 4, 9, 16, 25]
```

Quando si utilizzano **map** e **filter**, tornano particolarmente utili le funzioni anonime, che ci permettono di scrivere in maniera più compatta. Ad esempio, possiamo riscrivere quanto visto sopra come segue:

```
l2 = list(map(lambda x: x**2, l1))
print(l2)
```

```
[0, 1, 4, 9, 16, 25]
```

Filter permette di selezionare un sottoinsieme degli elementi di una lista sulla base di una condizione. La condizione viene specificata passando una funzione che prende in input l'elemento della lista e restituisce un booleano:

```
print(list(filter(lambda x: x%2==0,l1))) #filtra solo i numeri pari
```

```
[0, 2, 4]
```

2.16. Programmazione Orientata agli Oggetti - Definizione di Classi

La programmazione orientata agli oggetti in Python è intuitiva. Le principali differenze rispetto ai più diffusi linguaggi di programmazione orientata agli oggetti sono le seguenti:

- non esistono i modificatori di visibilità (**private**, **protected** e **public**). Per convenzione, tutti i simboli privati vanno preceduti da "__";
- ogni metodo è una funzione con un argomento di default (**self**) che rappresenta lo stato dell'oggetto;
- il costruttore si chiama "__init__".

```
class Classe(object): #ereditiamo dalla classe standard "object"
    def __init__(self, x): #costruttore
        self.x=x #inserisco il valore x nello stato dell'oggetto

    def prt(self):
        print(self.x)

    def power(self,y=2):
        self.x =  self.x**y

c = Classe(3)
c.prt()
c.power(3)
c.prt()
c.power()
c.prt()
```

```
3
27
729
```

Per estendere una classe, si fa come segue:

```
class Classe2(Classe): #derivo la classe "Classe" ed eredito metodi e
    def __init__(self,x):
        super(Classe2, self).__init__(x) #chiamo il costruttore della
    def prt(self): #ridefinisco il metodo prt
        print("yeah",self.x)

c2 = Classe2(2)
c2.power()
c2.prt()
```

```
yeah 4
```

2.17. Duck Typing

Per identificare i tipi dei dati, Python segue il principio del duck typing. Secondo tale principio, i tipi sono definiti utilizzando il *duck test*:

> If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

Ciò significa che i tipi sono definiti in relazione alle operazioni che possono essere eseguite su di essi. Vediamo un esempio (preso da Wikipedia).

```
class Sparrow(object):
    def fly(self):
        print("Sparrow flying")

class Airplane(object):
    def fly(self):
        print("Airplane flying")

class Whale(object):
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

sparrow = Sparrow()
airplane = Airplane()
whale = Whale()

try:
    lift_off(sparrow)
    lift_off(airplane)
    lift_off(whale) #Errore, il "tipo" di questo oggetto non permette
                    #Secondo il duck test, questo tipo è incompatibil
except AttributeError as e:
    print("Error:",e)
```

```
Sparrow flying
Airplane flying
Error: 'Whale' object has no attribute 'fly'
```

2.18. Eccezioni

In maniera simile a molti linguaggi moderni, Python supporta l'uso delle eccezioni. E' possibile catturare una eccezione con il costrutto `try - except`:

```
try:
    5/0
except:
    print("Houston, abbiamo un problema!")
```

```
Houston, abbiamo un problema!
```

In Python, le eccezioni sono tipizzate. Possiamo decidere quali tipi di eccezioni catturare come segue:

```
try:
    5/0
except ZeroDivisionError:
    print("Houston, abbiamo un problema!")
```

```
Houston, abbiamo un problema!
```

Possiamo avere accesso all'eccezione scatenata (ad esempio ottenere maggiori informazioni) come segue:

```
try:
    5/0
except ZeroDivisionError as e:
    print("Houston, abbiamo un problema!")
    print(e)
```

```
Houston, abbiamo un problema!
division by zero
```

E' possibile lanciare una eccezione mediante `raise`:

```
def div(x,y):
    if y==0:
        raise ZeroDivisionError() #lancia una eccezione
    else:
        return x/y
div(5,2)
div(5,0)
```

Possiamo definire nuove eccezioni estendendo la classe `Exception`:

```
class MyException(Exception):
    def __init__(self, message):
        self.message = message

raise MyException("Exception!")
```

Un modo veloce e comodo per lanciare una eccezione (un AssertionError nello specifico) quando qualcosa va male, è utilizzare `assert`, che prende in input un booleano e, opzionalmente, un messaggio di errore. Il booleano va posto uguale a `False` se qualcosa è andato storto. Vediamo un esempio:

```
def div(x,y):
    assert y!=0, "Cannot divide by zero!"
    return x/y

div(5,2)
div(5,0)
```

2.19. Definizione di Moduli

Quando si costruiscono programmi complessi, può essere utile raggruppare le definizioni di funzione e classi in moduli. Il modo più semplice di definire un modulo in Python consiste nell'inserire le definizioni all'interno di un file apposito `modulo.py`. Le definizioni potranno poi essere importati mediante la sintassi `from modulo import funzione`, a patto che il file che richiama le funzioni e quello che definisce il modulo si trovino nella stessa cartella. Vediamo un esempio:

```
#file modulo.py
def mysum(a,b):
    return a+b

def myprod(a,b):
    return a*b
```

```
#file main.py (stessa cartella di modulo.py)
from modulo import mysum, myprod
print(mysum(2,3)) #5
print(myprod(2,3)) #6
```

Altre informazioni su usi più avanzati di moduli e pacchetti possono essere reperite qui: https://docs.python.org/3/tutorial/modules.html.

```
import numpy as np #la notazione "as" ci permette di referenziare il
```

```
l = [[1,2,3],[4,5,2],[1,8,3]] #una lista contenente tre liste
print("List of lists:",l) #viene visualizzata così come l'abbiamo def
a = np.array(l) #costruisco un array di numpy a partire dalla lista d
print("Numpy array:\n",a) #ogni lista interna viene identificata come
print("Numpy array from tuple:\n",np.array(((1,2,3),(4,5,6)))) #posso
```

```
List of lists: [[1, 2, 3], [4, 5, 2], [1, 8, 3]]
Numpy array:
 [[1 2 3]
 [4 5 2]
 [1 8 3]]
Numpy array from tuple:
 [[1 2 3]
 [4 5 6]]
```

Ogni array di numpy ha una proprietà *shape* che ci permette di determinare il numero di dimensioni della struttura:

```
print(a.shape) #si tratta di una matrice 3 x 3
```

```
(3, 3)
```

Vediamo qualche altro esempio

```
array = np.array([1,2,3,4])
matrice = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
tensore = np.array([[[1,2,3,4],['a','b','c','d']],[[5,4,2,3],['a','b'
print('Array:',array, array.shape) #array monodimensionale, avrà una
print('Matrix:\n',matrice, matrice.shape)
print('matrix:\n',tensore, tensore.shape) #tensore, avrà due dimensio
```

```
Array: [1 2 3 4] (4,)
Matrix:
 [[1 2 3 4]
 [5 4 2 3]
 [7 5 3 2]
 [0 2 3 1]] (4, 4)
matrix:
 [[['1' '2' '3' '4']
  ['a' 'b' 'c' 'd']]

 [['5' '4' '2' '3']
  ['a' 'b' 'c' 'd']]

 [['7' '5' '3' '2']
  ['a' 'b' 'c' 'd']]

 [['0' '2' '3' '1']
  ['a' 'b' 'c' 'd']]] (4, 2, 4)
```

Vediamo qualche operazione tra numpy array:

```
a1 = np.array([1,2,3,4])
a2 = np.array([4,3,8,1])
print("Sum:",a1+a2) #somma tra vettori
print("Elementwise multiplication:",a1*a2) #moltiplicazione tra eleme
print("Power of two:",a1**2) #quadrato degli elementi
print("Elementwise power:",a1**a2) #elevamento a potenza elemento per
print("Vector product:",a1.dot(a2)) #prodotto vettoriale
print("Minimum:",a1.min()) #minimo dell'array
print("Maximum:",a1.max()) #massimo dell'array
print("Sum:",a2.sum()) #somma di tutti i valori dell'array
print("Product:",a2.prod()) #prodotto di tutti i valori dell'array
print("Mean:",a1.mean()) #media di tutti i valori dell'array
```

```
Sum: [ 5  5 11  5]
Elementwise multiplication: [ 4  6 24  4]
Power of two: [ 1  4  9 16]
Elementwise power: [   1    8 6561    4]
Vector product: 38
Minimum: 1
Maximum: 4
Sum: 16
Product: 96
Mean: 2.5
```

Operazioni tra matrici:

```
m1 = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
m2 = np.array([[8,2,1,4],[0,4,6,1],[4,4,2,0],[0,1,8,6]])

print("Sum:",m1+m2) #somma tra matrici
print("Elementwise product:\n",m1*m2) #prodotto elemento per elemento
print("Power of two:\n",m1**2) #quadrato degli elementi
print("Elementwise power:\n",m1**m2) #elevamento a potenza elemento p
print("Matrix multiplication:\n",m1.dot(m2)) #prodotto matriciale
print("Minimum:",m1.min()) #minimo
print("Maximum:",m1.max()) #massimo
print("Minimum along columns:",m1.min(0)) #minimo per colonne
print("Minimum along rows:",m1.min(1)) #minimo per righe
print("Sum:",m1.sum()) #somma dei valori
print("Mean:",m1.mean()) #valore medio
print("Diagonal:",m1.diagonal()) #diagonale principale della matrice
print("Transposed:\n",m1.T) #matrice trasposta
```

```
Sum: [[ 9  4  4  8]
 [ 5  8  8  4]
 [11  9  5  2]
 [ 0  3 11  7]]
Elementwise product:
 [[ 8  4  3 16]
 [ 0 16 12  3]
 [28 20  6  0]
 [ 0  2 24  6]]
Power of two:
 [[ 1  4  9 16]
 [25 16  4  9]
 [49 25  9  4]
 [ 0  4  9  1]]
Elementwise power:
 [[   1    4    3  256]
 [   1  256   64    3]
 [2401  625    9    1]
 [   1    2 6561    1]]
Matrix multiplication:
 [[20 26 51 30]
 [48 37 57 42]
 [68 48 59 45]
 [12 21 26  8]]
Minimum: 0
Maximum: 7
Minimum along columns: [0 2 2 1]
Minimum along rows: [1 2 2 0]
Sum: 47
Mean: 2.9375
Diagonal: [1 4 3 1]
Transposed:
 [[1 5 7 0]
 [2 4 5 2]
 [3 2 3 3]
 [4 3 2 1]]
```

2.20. Esercizi

> 👷 Esercizio 1
>
> Definire la lista `[1,8,2,6,15,21,76,22,0,111,23,12,24]`, dunque:
>
> - Stampare il primo numero della lista;
> - Stampare l'ultimo numero della lista;

- Stampare la somma dei numeri con indici dispari (e.g., 1,3,5,...) nella lista;
- Stampare la lista ordinata in senso inverso;
- Stampare la media dei numeri contenuti nella lista.

👨‍💻 Esercizio 2

Si definisca un dizionario `mesi` che mappi i nomi dei mesi nei loro corrispettivi numerici. Ad esempio, il risultato di:

```
print mesi['Gennaio']
```

deve essere

```
1
```

👨‍💻 Esercizio 3

Si considerino le seguenti liste:

```
l1 = [1,2,3]
l2 = [4,5,6]
l3 = [5,2,6]
```

Si combinino un ciclo for, **zip** e **enumerate** per ottenere il seguente output:

```
0 -> 10
1 -> 9
2 -> 15
```

👨‍💻 Esercizio 4

Si ripeta l'esercizio **2** utilizzando la comprensione di dizionari. A tale scopo, si definisca prima la lista `['Gennaio','Febbraio','Marzo','Aprile','Maggio','Giugno', 'Luglio','Agosto','Settembre','Ottobre','Novembre','Dicembre']`.

Si costruisca dunque il dizionario desiderato utilizzando la comprensione di dizionari e la funzione `enum`

👨‍💻 Esercizio 5

Date le variabili:

```
obj='triangolo'
area=21.167822
```

stampare le stringhe:

- L'area del triangolo è 21.16
- 21.1678 è l'area del triangolo

Utilizzare la formattazione di stringhe per ottenere il risultato.

👨‍💻 Esercizio 6

Scrivere una funzione che estragga il dominio da un indirizzo email. Ad esempio, se l'indirizzo è "furnari@dmi.unict.it", la funzione deve estrarre "dmi.unict.it".

Previous
1. Introduzione ai laboratori e Installazione dell'Ambiente di Lavoro

Next
3. Introduction to Data Analysis and Key Concepts