Impazziamo insieme con Python

A cura di: Martina L. Bentivegna, Maria C. Ferlito, Cristina M. Rao, Agnese Spinella.

SOMMARIO:

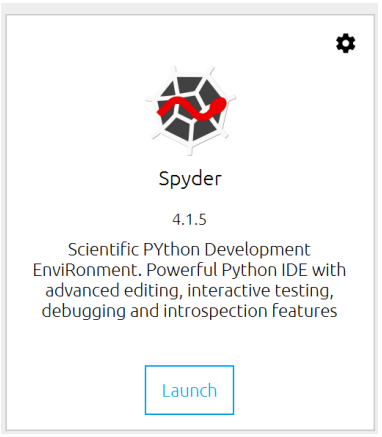
PARTIAMO DALLE BASI: download

LE NOZIONI
- Parameter Values di sort:
- Parameter Values
- TUPLE
- METODI
- ISTRUZIONI IF/ELSE/ELIF
- CICLO WHILE
- CICLO FOR
- Ordine operazioni su Python:
- FUNZIONI
- FUNZIONI MATEMATICHE
- OPERATORI
- DIZIONARI

PARTIAMO DALLE BASI:

Prima di iniziare con la guida vera e propria, è necessario avere un programma che ci faccia sperimentare con il linguaggio. Il programma che utilizzeremo per svolgere i vari esercizi è Spyder. Questo programma è facilmente scaricabile tramite Anaconda, applicazione che permette di scaricare e distribuire vari linguaggi di programmazione che sfruttano Python in modo diverso.

Una volta installato ed avviato Anaconda, dovrete cliccare su Launch Spyder.



Da qui potremmo cominciare a impazzire con questo bellissimo linguaggio di programmazione!

LE NOZIONI

LIBRERIA: è una collezione di moduli, che contengono le istruzioni per fornire funzioni e script già pronti. Per usarla serve la parola chiave *import*.

Esempio: import math

FUNZIONE: F(x) è una serie di istruzioni raggruppate sotto il nome della funzione scelto da noi (alla funzione possiamo affibbiare qualsiasi nome, persino "pippo"). La F si individua tra una coppia di (). Non c'è un limite di elementi che si possono inserire tra le parentesi di una funzione.

DEF: parola chiave per definire una funzione. Ad esempio, "def pippo():" servirà a programmare la funzione "pippo"

VALORE: unità di dati che un programma elabora. Ci sono diversi tipi di valore: numero intero, floating point (numero a virgola mobile), stringa, booleano (ossia, un valore che è o true o false)

STRINGA: sequenza di caratteri alfanumerici chiusi da virgolette (“”) o apici (‘’). Le stringhe sono immutabili, ovvero non possono essere modificate, occorre quindi sovrascrivere creando una nuova stringa.

Esempio: “Ho sonno2”

VARIABILE: fa riferimento ad un valore.

ISTRUZIONE: assegna alla variabile un valore. L’etichetta cioè il nome della variabile può contenere caratteri alfanumerici e underscore _ (no caratteri speciali, no spazi, no simboli).

RETURN: restituisce qualcosa in output.

Esempio:
```
def somma (x,y):  # all’interno delle () abbiamo gli argomenti della funzione.
    z = x+y
    return z
```

SLICING: Tecnica che permette di prendere sottostringhe della stringa in questione

Esempio: a = “linguistica”

a[2:5]

“ngu”

Sostanzialmente si va a trovare un intervallo che va dal 2 al 5. Il 2 è contenuto, il 5 no: l’ultimo carattere è sempre escluso!

Inoltre, si può omettere un estremo per indicare che si deve considerare il tutto dall’inizio [:5] = “lingu”, o fino alla fine [2:] = “nguistica”.

[:] = ci restituisce l’intera stringa.

E se utilizzo numeri negativi?

Il conteggio inizia da destra.

Esempi: a[:-1] = “linguistic” a [-5:-3] = ‘st’

l (-11) i (-10) n(-9) g(-8) u(7) i(-6) **s(-5)** t(-4) **i(-3)** c(-2) **a(-1)**

Inoltre, nello slicing è possibile specificare un passo, ad esempio:

```
>>>anni=[1,2,3,4,5,6,7,8,9,10]
>>>p=anni[1:10:2]      # si legge da 1 a 10 a passo di 2
print(p)
```

Si stamperanno gli elementi dall’indice 1, quindi dal numero 2, fino all’indice 9 a passi di 2. Si avrà cioè il seguente output [2, 4, 6, 8, 10].

Per ottenere un ordine inverso posso indicare semplicemente lo step di -1, come da esempio sotto:

```
>>> anni=[1,2,3,4,5,6,7,8,9,10]
>>>p=anni[::-1]
>>>print(p)
```

Stamperà quindi [10, 9, 8, 7, 6, 5, 4, 3, 2, 1].

Posso indicare anche lo step di -2 per stampare i numeri [10, 8, 6, 4, 2].

```
 anni=[1,2,3,4,5,6,7,8,9,10]
p=anni[-1:-10:-2]
print(p)
[10, 8, 6, 4, 2]
```

Quindi:
[x:x] stringa vuota
[:] tutto
[::1] leggi tutto
[::-1] leggi al contrario

Attenzione: Lo slicing potrà essere utilizzato nelle LISTE.

LISTE: come le stringhe, sono una sequenza di valori. Tuttavia, al contrario delle stringhe immutabili, le liste sono mutabili e sono racchiuse tra una coppia di []
Per le liste abbiamo opzioni particolari, quali:

SPLIT: metodo che divide la stringa in un insieme ordinato di parole trasformandola in una lista.

Esempio: s = "nel mezzo del cammin di nostra vita"
s.split()
["nel", "mezzo", "del", "cammin", "di", "nostra", "vita"]

#Attenzione: se non si inserisce nessun parametro all'interno del metodo split, implicitamente il metodo prende come parametro lo spazio. Altrimenti:

```
>>> s.split("e")
>>> ['n', 'l m', 'zzo d', 'l cammin di nostra vita']
```

JOIN: metodo che restituisce una stringa unendo tutti gli elementi di un iterabile (lista, stringa, tupla), separati da un separatore di stringa.

```
text = ['Python', 'is', 'a', 'fun', 'programming', 'language']

# unisce gli elementi di text con uno spazio.
print(' '.join(text))

# Output: Python is a fun programming language
```

APPEND: metodo che permette di aggiungere elementi al nostro inventario.

Esempio: inventario = ["torcia", "spada", "arco"]
inventario.append("frecce")
inventario = ["torcia", "spada", "arco", "frecce"]

#*append* aggiunge elementi solo alla fine

INSERT: metodo che permette di aggiungere elementi al nostro inventario (secondo argomento) in una specifica posizione (primo argomento).

Esempio: inventario = [“torcia”, “spada”, “arco”]
inventario.insert(2, “frecce”)
inventario = [“torcia”, “spada”,”frecce”, “arco”]

#2 è la posizione in cui voglio che venga inserito

REMOVE: metodo che permette di eliminare elementi dal nostro inventario.

Esempio: inventario = [“torcia”, “spada”, “arco”]
inventario.remove(“torcia”)
inventario = [“spada”, “arco”]

POP: metodo che ci permette di eliminare l’ultimo elemento dal nostro inventario. Inoltre, lo toglie dalla lista e lo restituisce in output. Se vogliamo essere più precisi, dobbiamo inserire la posizione.

Esempio: inventario = [“torcia”, “spada”, “arco”]
inventario.pop()
arco

fruits = [‘apple’, ‘banana’, ‘cherry’, ‘pear’]
fruits.pop(2)
‘cherry’
print (fruits)
[‘apple’, 'banana’, ‘pear’]

DEL: metodo che ci permette di eliminare elementi dal nostro inventario in base alla posizione

Esempio: inventario = [“torcia”, “spada”, “arco”]
del inventario[1]
inventario = [“torcia”, “arco”]

INDEX: metodo che ci permette di restituire la posizione di un elemento all’interno di una lista.

Esempio: a = [“a”, “b”, “c”, “d”, 2, 3]
a.index(d)
3

REVERSE: metodo che inverte gli elementi di una lista.

Esempio: a = [“a”, “b”, “c”, 1, 2, 3]
a.reverse()
a = [3, 2, 1, “c”, “b”, “a”]

#così facendo crea una nuova lista
#oppure utilizzare a[::-1] al posto di *reverse*. Questa operazione ci permette di mantenere la lista originaria

SORT: metodo che ci permette di ordinare secondo ordine alfabetico/numerico gli elementi della nostra adorata lista di merda (cit). Metodo non applicabile per liste che contengono stringhe + numeri/ booleani.
Esempio:
>>>t = [‘d’, ‘c’, ‘e’, ‘b’, ‘a’,’3’,’2’]
>>> t.sort()
>>> t
[‘2’,’3’,’a’, ‘b’, ‘c’, ‘d’, ‘e’]

È possibile utilizzare il metodo SORT anche se nella lista sono presenti sia valori interi numerici sia valori booleani.
Esempio:
>>> a=[4,3,True]
>>> a.sort()
>>> [True, 3, 4]
#True==1, invece False==0

Parameter Values di sort:

Parameter	Description
reverse	Opzionale. reverse=True ritornerà una lista decrescente. Di default è reverse=False
key	Opzionale. La funzione specifica i criteri della di sort

Esempio:
```
# La funzione ritorna la lunghezza del valore:
        def myFunc(e):
          return len(e)
cars = ['Ford', 'Mitsubishi', 'BMW', 'VW']
        cars.sort(key=myFunc)
print(cars)
['VW', 'BMW', 'Ford', 'Mitsubishi']          # mi ritornerà una lista ordinata secondo la lunghezza dei singoli elementi.
```

***RANGE*:** permette di generare liste particolari tramite intervalli di valori.
Esempio: range(5)

[0, 1, 2, 3, 4]

Range prevede 3 parametri: range*(start, stop, step*)

Parameter Values

Parameter	Description
start	Facoltativo. Indica la posizione di partenza. Di default parte da 0.
stop	Obbligatorio. Indica la posizione in cui si deve fermare (estremo destro non incluso
step	Facoltativo. Specifica l'incremento. Di default is 1

```
x = range(3, 6)
for n in x:
  print(n)
3
4
5
```

```
x = range(3, 20, 2)
for n in x:
  print(n)
3
5
7
9
11
13
15
17
19
```

TUPLE

Funziona come le lista ma si indica con (,)
t (1,2,3,4,5)
Le tuple sono un tipo di *sequenza* (come le **stringhe**), e supportano le operazioni comuni a tutte le sequenze, come *indexing*, *slicing*, contenimento, concatenazione, e ripetizione.
Le tuple sono **immutabili**, quindi una volta create non è possibile aggiungere, rimuovere, o modificare gli elementi.
I duplicati possono essere inseriti nella tuple, al contrario dei dizionari.
Nelle Tuple possiamo applicare queste funzioni e metodi:

LEN
MIN
MAX
COUNT

METODI

I metodi sono simili alle funzioni ma con due differenze:

- I metodi sono definiti all'interno della definizione di classe per rendere più esplicita la relazione tra la classe ed i metodi corrispondenti.

  Le classi sono un fantastico strumento che ci permette di raggruppare variabili e funzioni nei nostri programmi in maniera logica e riutilizzabile il che ci consente di gestire progetti anche di grosse dimensioni in maniera molto semplice. *(NDD il concetto verrà meglio compreso nel modulo B, non disperate se l'argomento sembra sanscrito)*

- La sintassi per invocare un metodo è diversa da quella usata per chiamare una funzione.
- I metodi si invocano con la classica notazione *oggetto.metodo (lista di parametri)*

Esempio:

```
parola = 'banana'
nuova_parola = parola.upper()
nuova_parola
'BANANA'
```

COUNT: ci restituisce quante volte un dato elemento è ripetuto.

Esempio: a = "linguistica"

```
a.count("i")
3
```

FIND: restituisce la **prima posizione** dell'occorrenza che ci serve.

Esempio: a = "linguistica"

```
a.find("i")
1
```

#il risultato è 1 perchè il conteggio parte da 0, quindi
l(0)**i(1)**n(2)g(3)u(4)i(5)s(6)t(7)i(8)c(9)a(10)
La funzione accetta anche un secondo parametro che indicherebbe il punto a partire dal quale occorre iniziare la ricerca.

Esempio: a.find('i', 2)
5
l(0)i(1)n(2)g(3)u(4)**i(5)**s(6)t(7)i(8)c(9)a(10)

Operazione inversa di FIND: se io voglio sapere cosa ci sta in una data posizione
Esempio: a = "linguistica"
a[3]
"g"
Invece, RFIND fa partire la ricerca da dx, quindi dalla fine

ENDSWITH: ci dice come finisce una stringa restituendo valori booleani
Esempio: a = "linguistica"
a.endswith("a")
True

ISDIGIT: ci dice se una stringa è formata solo da numeri, restituendo valori booleani.
Esempio: "abc123".isdigit()
False #perchè ci sono anche numeri

"abc".isdigit()
True #perchè ha solo caratteri
Cosa succede se inserisco l'argomento dentro le parentesi di ISDIGIT? Darebbe errore perchè non vuole PARAMETRI.

CAPITALIZE: trasforma in maiuscolo la prima lettera della stringa.
Esempio: a = "linguistica"
a.capitalize()
"Linguistica"

#non prevede parametri

LOWER: converte intera stringa in carattere minuscolo.
Esempio: a = "Linguistica"
a.lower()
"linguistica"
UPPER: converte intera stringa in carattere maiuscolo . Il contrario di *lower*
Esempio: a = "Linguistica"
a.upper()
"LINGUISTICA"

E se io in "linguistica", piuttosto che avere la prima lettera maiuscola voglio trasformarne un'altra? Tipo "lingUistica"

Applicando lo slicing è possibile scegliere i caratteri da modificare.
Esempio: a= ‘linguistica’
a[3].upper()
‘lingUistica’

Oppure: si può utilizzare la funzione Replace. Un esempio può essere:
>>> a.replace("u", "U")
>>> 'lingUistica'

REPLACE: prende in input due argomenti. Il primo argomento è quello che si vuole sostituire, il secondo argomento è quello che va a rimpiazzare il primo.
Esempio: a=”linguistica”
a.replace(“i”,”x”)
“lxnguxstxca”

Fine della parte dei metodi:
“in” e impiego di file testuali

‘y’ **IN** x: verifica se il carattere è presente nella stringa. Ci restituisce un valore booleano.
Esempio : ‘f’ in ‘linguistica’
false

FILE DI TESTO: è possibile caricare un file di testo (ossia, dal formato .txt) all’interno di una variabile tramite la funzione *open* all’interno della quale inserire una stringa
Esempio:f= open(“tindari.txt”)
#identifica il nome del file per poi inserirlo nella variabile che abbiamo scelto
Per leggere il file che abbiamo caricato utilizzare la funzione *read*
Esempio: f.read() mettendola sempre all’interno della variabile che abbiamo scelto
noi -> testo = f.read()

Per chiudere il file aperto usare la funzione *close*
Esempio: testo = f.close()
Per visualizzare il testo del file che abbiamo aperto usare la funzione *print*
Esempio: print(testo)
Il testo è modificabile. Se, ad esempio, vogliamo eliminare i punti usare la funzione *replace*
Esempio: testo.replace(“.”, “ “)

ISTRUZIONI IF/ELSE/ELIF

IF/ELSE: If (espressione booleana):

```
            blocco 1
        else:
            blocco 2
if condizione:
   # gruppo di istruzioni da eseguire
   # se la condizione è vera
else:
   # gruppo di istruzioni da eseguire
   # se la condizione è falsa
```

ELIF: “ se le condizioni precedenti non sono vere, allora prova questa condizione”.

```
a = 33
b = 33
if b > a:
        print("b is greater than a")
elif a == b:
        print("a and b are equal")

if condizione1:
   # gruppo di istruzioni eseguite
   # se la condizione1 è vera
elif condizione2:
   # gruppo di istruzioni eseguite
   # se la condizione2 è vera
elif condizioneN:
   # gruppo di istruzioni eseguite
   # se la condizioneN è vera
else:
   # gruppo di istruzioni eseguite
   # se tutte le condizioni sono false

if x < y:
        print('x è minore di y')
elif x > y:
        print('x è maggiore di y')
else:
        print('x e y sono uguali')
```

Stiamo ordinando alla consolle: se ciò che è scritto nell’espressione booleana è vero, esegui il blocco 1; altrimenti, esegui blocco 2. Il blocco si considera terminato quando si ritorna allo stesso livello dell’indentazione di If. Se non vogliamo continuare nel caso in cui il blocco 1 è falso, non è necessario inserire l’else.

Esempio:

```
i=3
j=i+1
if (j==4):
        print("il valore della j è uguale a 4")
else (j!=4):
        print("il valore della j è diverso da 4")
```

Differenze tra if / for / else. Quando programmiamo, il codice viene eseguito in maniera sequenziale. Ci sono due strumenti che impediscono le sequenze: le strutture di controllo, come IF (significa "se"), associato ad un "Else", altrimenti. Nell'IF, dobbiamo mettere una condizione: se è vera, entro nell'IF ed eseguo le istruzioni. Se è falsa, entro nell'ELSE, ovvero un altro blocco di istruzioni. La condizione deve essere un qualcosa che dobbiamo valutare se vera/falsa (valore booleano = True/false). Ma per capire cosa ci restituisce vero o falso? Usiamo degli operatori di confronto. Chiedo a Python se **è vera/falsa/ ==** (mi raccomando, non solo = perché significa assegnare), **!= (diverso), >=, <=.**

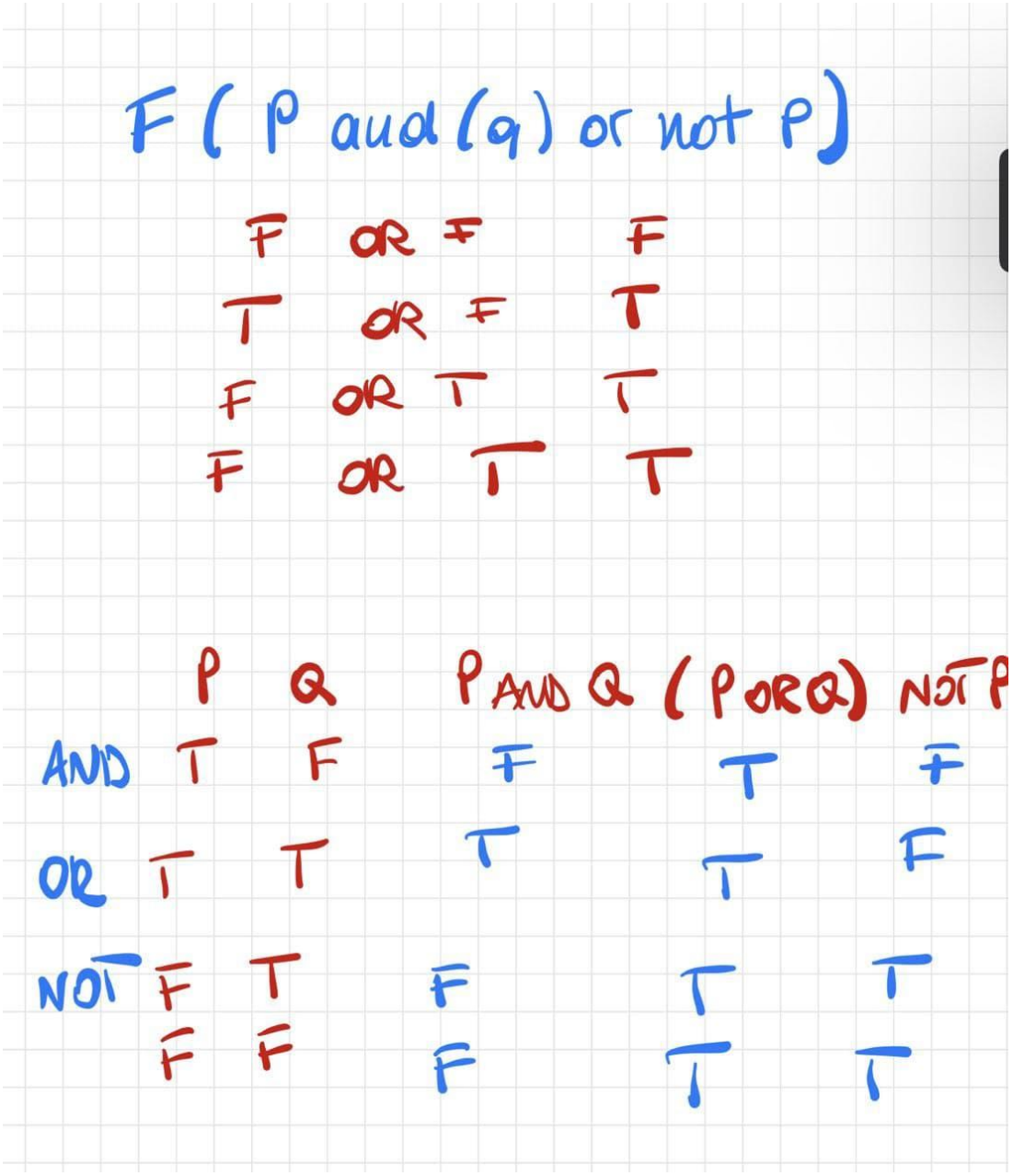
Oltre a questo, se io voglio usare delle condizioni più complicate, sostanzialmente devo usare i valori booleani, fare operazioni tra booleani. Gli operatori booleani sono: **and/or/not**. Mettono insieme i booleani, fanno delle operazioni. Mettiamo ad esempio che abbiamo due predicati: p e q. Mettiamo caso che P sia vero e Q sia falso. Possono essere anche tutti e due sono veri/falsi/, veri- falsi e al contrario (tavola della verità).

Se uso P and Q = devono accadere tutti e due contemporaneamente, è come una promessa (se io voglio valutare "oggi c'è il sole" and "i maiali volano", risulterà falsa, perché non accadono contemporaneamente).

Se uso P or Q = basta che ce ne sia uno vero (Decido di portare l'ombrello con me "se piove" o "se i maiali volano", una è vera e quindi risulta TRUE).

Se uso P not Q = fa il contrario.

Esempio: if ((P and Q) or not P).



Operatore IN = restituisce un booleano. Se scriviamo l[3,7,9], se scriviamo 4 in l, darà F. Se scriviamo 7 in l, darà invece T. Può essere utilizzato in tuple, liste, dizionari ecc. Esempio: ho una stringa s = “lorenzo”. Se scriviamo ‘o’ in s il risultato sarà True. Se scriviamo s = list(s), usando la funzione list, diverrà una lista. E la funzione in funzionerà anche allora

CICLO WHILE

```
while (espressione booleana): # finchè l'espressione booleana è vera lo esegue.
    blocco di istruzioni
```

rispetto ad if, ripete sempre lo stesso blocco finché non è falso.
Meno efficace, permette di far variare il valore di una variabile all'interno di un determinato range di valori permette di eseguire un blocco di istruzione se risulta essere vera.
Il ciclo While itera fintanto che la condizione è vera:

```
>>> # rimuovi e printa numeri da seq finchè ne rimangono solo 3
>>> seq = [10, 20, 30, 40, 50, 60]
>>> while len(seq) > 3:
...     print(seq.pop())
...
60
50
40
>>> seq
[10, 20, 30]
```

Possiamo notare che:

- il ciclo while è introdotto dalla keyword while, seguita da una condizione (len(seq) > 3) e dai due punti (:);
- dopo i due punti è presente un blocco di codice indentato (che può anche essere formato da più righe);
- il ciclo while esegue il blocco di codice fintanto che la condizione è vera;
- in questo caso rimuove e stampa gli elementi di seq fintanto che in seq ci sono più di 3 elementi;
- una volta che la sequenza è rimasta con solo 3 elementi, la condizione len(seq) > 3 diventa falsa e il ciclo termina.

```
>>> #Prendere in input 10 numeri e sommarli.
>>>#Usiamo una variabile contatore, che chiamiamo ad esempio i ed inizializziamo a 0.
>>>#Dopo, per ogni numero inserito, la incrementiamo di 1 (i=i+1).
>>>#Dunque il ciclo dovrà continuare finché i sarà minore o uguale a 10 (da 0 a 10 sono 11 numeri).
>>>#Allora la nostra condizione sarà questa: i<=10.
>>>#Come prima condizione stabilire che il contatore è uguale a 0 (il primo numero da prendere in considerazione).
i=0
while i<=10:
        i=i+1
print(i)                                #il risultato sarà 11 perchè conteggerà da 0 a 10.
```

CICLO FOR

Il ciclo FOR ci permette di iterare su tutti gli elementi iterabili ed eseguire un determinato blocco.
Il ciclo for ci permette di iterare su tutti gli elementi di un iterabile ed eseguire un determinato blocco di codice. Un iterabile è un qualsiasi oggetto in grado di restituire tutti gli elementi uno dopo l'altro, come ad esempio liste, tuple, set, dizionari (restituiscono le chiavi), ecc.
Esempio:

```
>>> # stampa il quadrato di ogni numero di seq
>>> seq = [1, 2, 3, 4, 5]
>>> for n in seq:
...     print('Il quadrato di', n, 'è', n**2)
...
Il quadrato di 1 è 1
Il quadrato di 2 è 4
Il quadrato di 3 è 9
Il quadrato di 4 è 16
Il quadrato di 5 è 25
```

```
prefissi = 'JKLMNOPQ'
suffisso = 'ack'
for lettera in prefissi:
print(lettera + suffisso)
```

Il risultato del programma è:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Perché prende solo la prima lettera? Perché il ciclo itera in tutta la stringa.

Ricordate che per risolvere un qualunque esercizio è meglio applicare la la strategia DIVIDI ET IMPERA, dividete il problema in tante piccole parti.
Esempio:

```
def e01(s): #s sta per stringa, la prende come parametro
```

voc = "aeiou"

for c in voc: #fa un ciclo per ogni lettera che vede nella stringa voc

if (c in s): #ha cercato tutte le vocali, al posto di c possiamo dare qualsiasi nome

return True #ma solamente se ci sono tutte le vocali

return False #in caso niente

print (e01("lorenzo"))

Spiegazione: Se usassimo count, prenderebbe le singole vocali. Se scrivessimo tutte le vocali, cercherebbe “aeiou”. Bisogna prima contarle con un ciclo, e poi vedere se le vocali contate col ciclo nella stringa si trovano nella stringa data in input. Il ciclo, data la stringa, scorre ogni lettera (può essere interrogata tramite indici). Potremmo scrivere anche “for i in range(len(voc))”, la i però sarebbe un numero ma non un carattere. A questo punto dovremmo scrivere voc[i], e sarebbe uguale ad a.

Ordine operazioni su Python:

Messi in ordine di precedenza.
PEMDAS : acronimo per ricordare ordine.

- operazioni dentro le parentesi. Es: 2 * (3-1) fa 4,
- elevamento a potenza. Es: 1 + 2**3 fa 9, e non 27
- Moltiplicazione e Divisione hanno la precedenza su Addizione e Sottrazione. Es: 2*3-1 fa 5, e non 4, e 6+4/2 fa 8, e non 5.
- le operazioni di uguale priorità vengono valutati da sinistra verso destra.

Operazioni sulle stringhe.

'2'-'1' 'uova'/'facili' 'terzo'*'una magia'
Queste operazioni non possono essere effettuate sulle stringhe, ma ci sono due eccezioni: + e *.

+ esegue il concatenamento, unisce due stringhe collegandole ai due estremi.

```
>>> primo = 'bagno'
>>> secondo = 'schiuma'
>>> primo + secondo
bagnoschiuma
```

* esegue la ripetizione sulle stringhe.
'Spam'*3 dà 'SpamSpamSpam'

FUNZIONI

***INT*:** dato un valore, lo converte se possibile in un numero intero.

```
>>> int('32')
        32
```

Può convertire anche i numeri con la virgola, ma non li arrotonda, bensì li tronca.

```
>>> int(3.99999)
        3
```

FLOAT: converte i numeri interi e stringhe in numeri con virgola mobile.

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

***STR*:** converte l'argomento in una stringa

```
>>> str(32)
'32'
```

MIN: Riporta il valore più basso in una sequenza di numeri o stringhe.

```
nums = [ 1,8,2,23,7,-4,18,23,42,37,2]
min(nums)
-4
```

***MAX*:** riparta il valore più alto presente in una sequenza di numeri o stringhe.

```
nums =  [ 1,8,2,23,7,-4,18,23,42,37,2]
max ( nums)
42
```

***TYPE*:** restituisce il valore di una dato elemento.

```
a = "cane44"
type(a)
str
```

LEN: restituisce la lunghezza.

Esempio: a = "linguistica"

```
len(a)
11
```

FUNZIONI MATEMATICHE

Un modulo è un file che contiene una raccolta di funzioni correlate.

Prima di poter usare un modulo lo dobbiamo importare, con l'istruzione IMPORT.
>>> import math
Questa istruzione crea un oggetto modulo chiamato math. L'oggetto modulo contiene le funzioni e le variabili definite all'interno del modulo stesso.
Per accedere a una funzione del modulo, dovete specificare, nell'ordine, il nome del modulo e il nome della funzione, separati da un punto.
MATH: contiene le operazioni matematiche complesse, come il p greco etc.

OPERATORI

x<y : x è minore di y
x<=y: x è minore o uguale a y
x>y: x è maggiore di y
x>=y: x è maggiore o uguale a y
x==y: comparazione tra le due variabili (vero/falso)
x!=y: x è diverso da y

COMPOSIZIONE
Consiste nella possibilità di inserire delle funzioni all'interno della funzione stessa. Prima vengono risolte quelle più interne, una volta ottenuto il risultato di queste funzioni, viene risolta la più esterna.

Esempio:
- 1 istruzione e nessuna variabile:
  print (math.floor (math.sqrt (45)))

- 3 istruzioni e due variabili:
  a = math.sqrt(45)
  b = math.floor(a)
  print (b)

Queste due funzioni si equivalgono.

DIZIONARI

Definiti all'interno di una coppia di {}. Sono mutabili. I dizionari sono formate da coppie di valori separate da due punti (:) Le coppie di valori sono chiave-valore
Esempio: d = {3:6, 2:4, 7:14, 4:8}
Il 3 è la chiave, il 6 è il valore. Possiamo trovare valori uguali ma mai chiavi uguali.
La stessa chiave non può avere due significati (valori) diversi, ma due chiavi diverse possono avere lo stesso significato (valore).

Se viene specificata una chiave inesistente, Python restituisce un KeyError. È però possibile usare l'operatore in (o not in) per verificare se una chiave è presente nel dizionario.

Metodi utilizzabili nei dizionari:
KEYS: metodo che ritorna le chiavi del dizionario. Non prende parametri.

```
person = {'name': 'Phill', 'age': 22, 'salary': 3500.0}
print(person.keys())

empty_dict = {}
print(empty_dict.keys())
```

ITEMS: metodo che ritorna la coppia chiave- valore sotto forma di tupla. Non prende argomenti.

```
# dizionario random
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

print(sales.items())
```

UPDATE*:* metodo che aggiunge elementi all'interno del dizionario, questi elementi possono provenire da altri dizionari o da tuple o stringhe.
Attenzione: Update aggiunge elementi all'interno del dizionario, ma se la chiave è già presente nel dizionario, aggiornerà solo il valore.

```
d = {1: "one", 2: "three"}
d1 = {2: "two"}

# updates the value of key 2
d.update(d1)

print(d)

d1 = {3: "three"}

# adds element with key 3
d.update(d1)

print(d)
```

```
{1: 'one', 2: 'two'}

{1: 'one', 2: 'two', 3: 'three'}
```

CLEAR*:*metodo che rimuove tutti gli elementi del dizionario. Non richiede argomenti.

```
d = {1: "one", 2: "two"}
```

```
d.clear()
print('d =', d)
d = {}
```

DEL (vedere la sezione dedicata alle liste).