



## **Corso di Apprendistato Python**

Mod. Python  
Docente:  
Tonino Petrulli



## **Perché abbiamo bisogno di liste?**

Può capitare di dover leggere, memorizzare, elaborare e infine stampare decine, forse centinaia, forse addirittura migliaia di numeri. Che cosa succede? Dovete creare una variabile separata per ogni valore? Dovrete passare lunghe ore a scrivere dichiarazioni come quella che segue?

```
var1 = int(input())  
var2 = int(input())  
var3 = int(input())  
var4 = int(input())  
var5 = int(input())  
var6 = int(input())
```

Finora abbiamo imparato a dichiarare variabili in grado di memorizzare esattamente un valore alla volta. Tali variabili sono talvolta chiamate scalari per analogia con la matematica. Tutte le variabili utilizzate finora sono in realtà scalari.

Pensate a quanto sarebbe comodo dichiarare una variabile che possa memorizzare più di un valore. Ad esempio, cento, o mille, o addirittura diecimila. Si tratterebbe sempre di una stessa variabile, ma molto ampia e capiente. Sembra interessante? Forse, ma come gestirebbe un contenitore così pieno di valori diversi? Come sceglierebbe quello di cui ha bisogno?

## **Perché abbiamo bisogno di liste?**

Creiamo una variabile chiamata `numeri`; le viene assegnato non un solo numero, ma un elenco di cinque valori (nota: l'elenco inizia con una parentesi quadra aperta e termina con una parentesi quadra chiusa; lo spazio tra le parentesi è riempito con cinque numeri separati da virgole).

```
numeri = [10, 5, 7, 2, 1]
```

`numeri` è una lista composta da cinque valori, tutti numeri. Possiamo anche dire che questa affermazione crea una lista di lunghezza pari a cinque (in quanto ci sono cinque elementi al suo interno).

Gli elementi di un elenco possono avere tipi diversi. Alcuni di essi possono essere numeri interi, altri numeri float e altri ancora possono essere elenchi.

Python ha adottato una convenzione secondo la quale gli elementi di un elenco sono sempre numerati a partire da zero. Ciò significa che l'elemento memorizzato all'inizio dell'elenco avrà il numero zero. Poiché nella nostra lista ci sono cinque elementi, all'ultimo di essi viene assegnato il numero quattro. Non dimenticatevi di questo.

## **Elenchi di indicizzazione**

Come si modifica il valore di un elemento scelto nell'elenco?

Assegniamo un nuovo valore di 111 al primo elemento dell'elenco. Lo facciamo in questo modo:

```
numeri = [10, 5, 7, 2, 1]  
print("Contenuto originale dell'elenco:", numeri) # Stampa del contenuto originale dell'elenco.
```

```
numeri[0] = 111  
print("Contenuto del nuovo elenco: ", numeri) # Contenuto dell'elenco attuale.
```

Se vogliamo che il valore del quinto elemento sia copiato nel secondo elemento:

```
numeri[1] = numeri[4] # Copia del valore del quinto elemento nel secondo.  
print("Nuovo contenuto dell'elenco:", numeri) # Stampa del contenuto dell'elenco attuale.
```

## Elenchi di indicizzazione

Il valore all'interno delle parentesi che seleziona un elemento dell'elenco è chiamato indice, mentre l'operazione di selezione di un elemento dall'elenco è nota come indicizzazione.

**Nota:** tutti gli indici utilizzati finora sono letterali. I loro valori sono fissati in fase di esecuzione, ma anche qualsiasi espressione può essere un indice. Questo apre molte possibilità.

l'elenco può anche essere stampato nel suo insieme, come in questo caso:

```
print(enumeri) # Stampa dell'intero elenco.
```

```
enumeri = [10, 5, 7, 2, 1]
```

```
print("Contenuto dell'elenco:", enumeri) # Stampa del contenuto dell'elenco
```

## La funzione len()

La lunghezza di un elenco può variare durante l'esecuzione. Nuovi elementi possono essere aggiunti all'elenco, mentre altri possono essere rimossi da esso. Ciò significa che l'elenco è un'entità molto dinamica.

Se si vuole controllare la lunghezza attuale dell'elenco, si può usare una funzione chiamata len() (il suo nome deriva da *length*).

La funzione prende come argomento il nome dell'elenco e restituisce il numero di elementi attualmente memorizzati all'interno dell'elenco (in altre parole, la lunghezza dell'elenco).

Osservate l'ultima riga di codice nell'editor, eseguite il programma e verificate quale valore verrà stampato nella console. Riuscite a indovinare?

```
numeri = [10, 5, 7, 2, 1]
print("Contenuto originale dell'elenco:", numeri) # Stampa del contenuto originale dell'elenco.
numeri[0] = 111
print("\nContenuto dell'elenco precedente:", numeri) # Stampa del contenuto dell'elenco precedente.
numeri[1] = numeri[4] # Copia del valore del quinto elemento nel secondo.
print("Contenuto dell'elenco precedente:", numeri) # Stampa del contenuto dell'elenco precedente.
print("Lunghezza elenco:", len(numeri)) # Stampa della lunghezza dell'elenco.
```

## Rimozione di elementi da un elenco

Qualsiasi elemento dell'elenco può essere rimosso in qualsiasi momento: ciò avviene con un'istruzione chiamata `del` (delete). **Nota:** si tratta di un'istruzione, non di una funzione.

È necessario indicare l'elemento da rimuovere: esso scomparirà dall'elenco e la lunghezza dell'elenco sarà ridotta di uno.

Guardate lo snippet qui sotto. Riuscite a indovinare quale output produrrà? Eseguite il programma nell'editor e verificate.

```
del numeri[1]  
print(len(numeri))  
print(numeri)
```

Non si può accedere a un elemento che non esiste: non si può né ottenere il suo valore né assegnargli un valore. Entrambe le istruzioni causeranno errori di runtime:

```
print(numeri[4])  
numeri[4] = 1
```

## **Gli indici negativi sono legali**

Può sembrare strano, ma gli indici negativi sono legali e possono essere molto utili. Un elemento con indice uguale a -1 è l'ultimo dell'elenco.

```
numeri = [10, 5, 7, 2, 1]  
print(numeri[-1])
```

Lo snippet di esempio produce 1. Eseguire il programma e verificare.

Allo stesso modo, l'elemento con indice uguale a -2 è quello che precede l'ultimo dell'elenco.

```
print(numeri[-2])
```

Lo snippet di esempio produce 2.

L'ultimo elemento accessibile della nostra lista è numeri[-4] (il primo): non cercate di andare oltre!



## LAB-20

**Tempo stimato:** 5 minuti

**Livello di difficoltà:** Molto facile

### Obiettivi

Familiarizzare lo studente con:  
utilizzando le istruzioni di base relative agli elenchi;  
creare e modificare gli elenchi.

### Scenario

C'era una volta un cappello. Il cappello non conteneva un coniglio, ma un elenco di cinque numeri: 1, 2, 3, 4 e 5.

Il vostro compito è quello di:

scrivere una riga di codice che chieda all'utente di sostituire il numero centrale dell'elenco con un numero intero inserito dall'utente (Passo 1)

scrivere una riga di codice che rimuova l'ultimo elemento dall'elenco (passo 2)

scrivere una riga di codice che stampi la lunghezza dell'elenco esistente (passo 3).

**hat\_list = [1, 2, 3, 4, 5] # Si tratta di un elenco esistente di numeri nascosti nel cappello.**

**# Passo 1: scrivere una riga di codice che richieda all'utente**

**# per sostituire il numero centrale con un numero intero inserito dall'utente.**

**# Passo 2: scrivere una riga di codice che rimuova l'ultimo elemento dall'elenco.**

**# Passo 3: scrivere una riga di codice che stampi la lunghezza dell'elenco esistente.**

**print(hat\_list)**

## Funzioni vs. metodi

Un metodo è un tipo specifico di funzione: si comporta come una funzione e ha l'aspetto di una funzione, ma si differenzia per il modo in cui agisce e per lo stile di invocazione.

Una funzione non appartiene a nessun dato: riceve dati, può crearne di nuovi e (generalmente) produce un risultato.

Un metodo fa tutte queste cose, ma è anche in grado di modificare lo stato di un'entità selezionata.

Un metodo è di proprietà dei dati per cui lavora, mentre una funzione è di proprietà dell'intero codice.

Ciò significa anche che l'invocazione di un metodo richiede uno specifico dato da cui il metodo viene invocato.

In generale, una tipica invocazione di funzione può assomigliare a questa:

risultato = funzione(arg)

La funzione prende un argomento, fa qualcosa e restituisce un risultato.

Una tipica invocazione di metodo si presenta di solito così:

risultato = data.method(arg)

**Nota:** il nome del metodo è preceduto dal nome del dato che possiede il metodo. Quindi si aggiunge un punto, seguito dal nome del metodo e da una coppia di parentesi che racchiudono gli argomenti.

Il metodo si comporta come una funzione, ma può fare qualcosa di più: può modificare lo stato interno dei dati da cui è stato invocato.

## **Aggiunta di elementi a un elenco: `append()` e `insert()`**

Un nuovo elemento può essere *incollato* alla fine dell'elenco esistente:

### **`list.append(valore)`**

Tale operazione viene eseguita da un metodo chiamato `append()`. Esso prende il valore del suo argomento e lo mette alla fine dell'elenco che possiede il metodo.

La lunghezza dell'elenco aumenta quindi di uno.

Il metodo `insert()` è un po' più intelligente: può aggiungere un nuovo elemento in qualsiasi punto dell'elenco, non solo alla fine.

### **`list.insert(posizione, valore)`**

Richiede due argomenti:

- la prima mostra la posizione richiesta per l'elemento da inserire; si noti che tutti gli elementi esistenti che occupano posizioni a destra del nuovo elemento (compreso quello nella posizione indicata) vengono spostati a destra, per fare spazio al nuovo elemento;
- il secondo è l'elemento da inserire.

## **Aggiunta di elementi a un elenco: `append()` e `insert()`**

Osservare il codice seguente. Osservate come vengono utilizzati i metodi `append()` e `insert()`. Prestate attenzione a ciò che accade dopo l'uso di `insert()`: il primo elemento è ora il secondo, il secondo il terzo e così via.

```
numbers = [111, 7, 2, 1]
print(len(numbers))
print(numbers)
###
numbers.append(4)
print(len(numbers))
print(numbers)
###
numbers.insert(0, 222)
print(len(numbers))
print(numbers)
```

Aggiungete il seguente snippet dopo l'ultima riga di codice nell'editor:

```
numbers.insert(1, 333)
```

## Utilizzo di elenchi

Il ciclo for ha una variante molto speciale che può elaborare gli elenchi in modo molto efficace: vediamola.

Supponiamo di voler calcolare la somma di tutti i valori memorizzati nell'elenco `my_list`.

È necessaria una variabile la cui somma verrà memorizzata e alla quale verrà inizialmente assegnato il valore 0. Il suo nome sarà `total`. (Nota: non la chiameremo `sum`, perché Python usa lo stesso nome per una delle sue funzioni incorporate, `sum()`. L'uso dello stesso nome è generalmente considerato una cattiva pratica). Quindi si aggiungono tutti gli elementi dell'elenco utilizzando il ciclo for. Date un'occhiata allo snippet:

```
my_list = [10, 1, 8, 3, 5]  
total = 0  
for i in range(len(my_list)):  
    total += my_list[i]  
print(total)
```

Commentiamo questo esempio:

- all'elenco viene assegnata una sequenza di cinque valori interi;
- la variabile `i` assume i valori 0, 1, 2, 3 e 4, quindi indicizza l'elenco, selezionando gli elementi successivi: il primo, il secondo, il terzo, il quarto e il quinto;
- ognuno di questi elementi viene sommato dall'operatore `+=` alla variabile `total`, fornendo il risultato finale alla fine del ciclo;

## La seconda faccia del ciclo for

Ma il ciclo for può fare molto di più. Può nascondere tutte le azioni legate all'indicizzazione dell'elenco e fornire tutti gli elementi dell'elenco in modo pratico.

Questo frammento modificato mostra come funziona:

```
my_list = [10, 1, 8, 3, 5]
total = 0
for i in my_list:
    total += i
print(total)
```

Cosa succede qui?

- l'istruzione for specifica la variabile utilizzata per sfogliare l'elenco (i qui) seguita dalla parola chiave in e dal nome dell'elenco da elaborare (my\_list qui)
- alla variabile i vengono assegnati i valori di tutti gli elementi successivi dell'elenco e il processo si ripete tante volte quanti sono gli elementi dell'elenco;
- significa che si usa la variabile i come copia dei valori degli elementi e non è necessario usare gli indici;
- la funzione len() non è necessaria nemmeno in questo caso.

## **Elenchi in azione**

Lasciamo da parte le liste per un breve momento e passiamo a una questione intrigante.

Immaginate di dover riordinare gli elementi di un elenco, cioè di invertire l'ordine degli elementi: il primo e il quinto elemento, così come il secondo e il quarto, verranno scambiati. Il terzo rimarrà inalterato.

Domanda: come si possono scambiare i valori di due variabili?

Date un'occhiata allo snippet:

```
variable_1 = 1  
variable_2 = 2  
variable_2 = variable_1  
variable_1 = variable_2
```

Se si fa una cosa del genere, si perderà il valore precedentemente memorizzato nella `variable_2`. Cambiare l'ordine delle assegnazioni non servirà a nulla. È necessaria una terza variabile che funga da memoria ausiliaria.

Ecco come si può fare:

```
variable_1 = 1  
variable_2 = 2  
auxiliary = variable_1  
variable_1 = variable_2  
variable_2 = auxiliary
```

## **Elenchi in azione**

Python offre un modo più comodo per effettuare lo scambio: date un'occhiata:

```
variabile_1 = 1  
variabile_2 = 2  
variabile_1, variabile_2 = variabile_2, variabile_1
```

Ora è possibile scambiare facilmente gli elementi dell'elenco per invertirne l'ordine:

```
my_list = [10, 1, 8, 3, 5]  
my_list[0], my_list[4] = my_list[4], my_list[0]  
my_list[1], my_list[3] = my_list[3], my_list[1]  
print(my_list)
```



## Elenchi in azione

E se dovessimo scambiare gli elementi in una lista lunga 100?

```
my_list = [10, 1, 8, 3, 5]
length = len(my_list)
```

```
for i in range(length // 2):
    my_list[i], my_list[length - i - 1] = my_list[length - i - 1], my_list[i]

print(my_list)
```

### Nota:

- abbiamo assegnato la variabile `length` con la lunghezza dell'elenco corrente (questo rende il nostro codice un po' più chiaro e più breve)
- abbiamo lanciato il ciclo `for` per percorrere il suo corpo di lunghezza `// 2` volte (questo funziona bene per liste di lunghezza sia pari che dispari, perché quando la lista contiene un numero dispari di elementi, quello centrale rimane intatto)
- abbiamo scambiato l'elemento  $i^{\text{esimo}}$  (dall'inizio dell'elenco) con quello con indice pari a  $(\text{lunghezza} - i - 1)$  (dalla fine dell'elenco); nel nostro esempio, per  $i$  uguale a 0 il  $(\text{lunghezza} - i - 1)$  dà 4; per  $i$  uguale a 1, dà 3 - questo è esattamente ciò di cui avevamo bisogno.

## **LAB-21**

**Tempo stimato:** 10-15 minuti

**Livello di difficoltà:** Facile

### **Obiettivi**

Familiarizzare lo studente con:  
creare e modificare semplici elenchi;  
utilizzando metodi per modificare gli elenchi.

### **Scenario**

I Beatles sono stati uno dei gruppi musicali più popolari degli anni '60 e la band più venduta della storia. Alcuni li considerano il gruppo più influente dell'era rock. Infatti, sono stati inclusi nella classifica delle 100 persone più influenti del XX secolo stilata dalla rivista *Time*.

La band subì numerosi cambi di formazione, culminati nel 1962 con la formazione di John Lennon, Paul McCartney, George Harrison e Richard Starkey (meglio conosciuto come Ringo Starr).

Scrivete un programma che rifletta questi cambiamenti e vi permetta di fare pratica con il concetto di lista. Il vostro compito è quello di:

## LAB-21

- passo 1: creare un elenco vuoto chiamato beatles;
- passo 2: utilizzare il metodo append() per aggiungere all'elenco i seguenti membri della band: John Lennon, Paul McCartney e George Harrison;
- passo 3: utilizzare il ciclo for e il metodo append() per chiedere all'utente di aggiungere all'elenco i seguenti membri della band: Stu Sutcliffe e Pete Best;
- passo 4: utilizzare l'istruzione del per rimuovere Stu Sutcliffe e Pete Best dall'elenco;
- passo 5: utilizzare il metodo insert() per aggiungere Ringo Starr all'inizio dell'elenco.

**# passo 1**

**print("Passo 1:", beatles)**

**# passo 2**

**print("Passo 2:", beatles)**

**# passo 3**

**print("Passo 3:", beatles)**

**# passo 4**

**print("Passo 4:", beatles)**

**# passo 5**

**print("Passo 5:", beatles)**

**# lista di test legth**

**print("The Fab", len(beatles))**

### **Esercizio 1**

Qual è l'output del seguente snippet?

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

```
lst.insert(1, 6)
```

```
del lst[0]
```

```
lst.append(1)
```

```
print(lst)
```

### **Esercizio 2**

Qual è l'output del seguente snippet?

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

```
lst_2 = []
```

```
add = 0
```

```
for number in lst:
```

```
    add += number
```

```
    lst_2.append(add)
```

```
print(lst_2)
```

### **Esercizio 3**

Cosa succede quando si esegue il seguente snippet?

```
lst = []  
del lst  
print(lst)
```

### **Esercizio 4**

Qual è l'output del seguente snippet?

```
lst = [1, [2, 3], 4]  
print(lst[1])  
print(len(lst))
```

## Il bubble sort

Ora che sapete destreggiarvi efficacemente tra gli elementi delle liste, è il momento di imparare a ordinarle. Finora sono stati inventati molti algoritmi di ordinamento, che differiscono molto per velocità e complessità. Vi mostreremo un algoritmo molto semplice, facile da capire, ma purtroppo non troppo efficiente. Viene utilizzato molto raramente, e certamente non per elenchi estesi e di grandi dimensioni.

Diciamo che un elenco può essere ordinato in due modi:

- crescente (o più precisamente - non decrescente) - se in ogni coppia di elementi adiacenti, il primo elemento non è maggiore del secondo;
- decrescente (o più precisamente - non crescente) - se in ogni coppia di elementi adiacenti, il primo elemento non è minore del secondo.

Nelle sezioni seguenti, ordineremo l'elenco in ordine crescente, in modo che i numeri siano ordinati dal più piccolo al più grande.

Ecco l'elenco:

8	10	6	2	4
---	----	---	---	---

## Il bubble sort

Cercheremo di utilizzare il seguente approccio: prenderemo il primo e il secondo elemento e li confronteremo; se determineremo che sono nell'ordine sbagliato (cioè, il primo è maggiore del secondo), li scambieremo; se il loro ordine è valido, non faremo nulla. Un'occhiata alla nostra lista conferma quest'ultima ipotesi: gli elementi 01 e 02 sono nell'ordine corretto, come in  $8 < 10$ .

Ora guardate il secondo e il terzo elemento. Sono nelle posizioni sbagliate. Dobbiamo scambiarli:

8	6	10	2	4
---	---	----	---	---

Andiamo oltre ed esaminiamo il terzo e il quarto elemento. Anche in questo caso, non è così che dovrebbe essere. Dobbiamo scambiarli:

8	6	2	10	4
---	---	---	----	---

Ora controlliamo il quarto e il quinto elemento. Sì, anche loro sono nelle posizioni sbagliate. Si verifica un altro scambio:

8	6	2	4	10
---	---	---	---	----

## Il bubble sort

Il primo passaggio dell'elenco è già terminato. Siamo ancora lontani dal finire il nostro lavoro, ma nel frattempo è successo qualcosa di curioso. L'elemento più grande, 10, è già finito alla fine dell'elenco. Si noti che questo è il posto desiderato. Tutti gli elementi rimanenti formano un pasticcio pittoresco, ma questo è già al suo posto. Ora, per un momento, provate a immaginare l'elenco in un modo leggermente diverso, cioè in questo modo:

10
4
2
6
8

Guardate: il 10 è in alto. Potremmo dire che è salito dal basso verso la superficie, proprio come la bolla in un bicchiere di champagne. Il metodo di ordinamento deriva il suo nome dalla stessa osservazione: si chiama ordinamento a bolle.



## Il bubble sort

Ora iniziamo con il secondo passaggio attraverso l'elenco. Osserviamo il primo e il secondo elemento: è necessario uno scambio:

6	8	2	4	10
---	---	---	---	----

È il momento del secondo e del terzo elemento: dobbiamo scambiare anche questi:

6	2	8	4	10
---	---	---	---	----

Ora il terzo e il quarto elemento e il secondo passaggio sono terminati, poiché l'8 è già in posizione:

6	2	4	8	10
---	---	---	---	----

## Il bubble sort

Si inizia subito il passaggio successivo. Osservare attentamente il primo e il secondo elemento: è necessario un altro scambio:

2	6	4	8	10
---	---	---	---	----

Ora il 6 deve essere posizionato. Scambiamo il secondo e il terzo elemento:

2	4	6	8	10
---	---	---	---	----

La lista è già ordinata. Non abbiamo più nulla da fare. Questo è esattamente ciò che vogliamo.

Come si può vedere, l'essenza di questo algoritmo è semplice: si confrontano gli elementi adiacenti e, scambiandone alcuni, si raggiunge l'obiettivo.

Codifichiamo in Python tutte le azioni eseguite durante un singolo passaggio attraverso l'elenco e poi consideriamo quanti passaggi sono necessari per eseguirli.

## Ordinamento di un elenco

Quanti passaggi sono necessari per ordinare l'intero elenco? Quante volte dovremmo fare il seguente ciclo for?

```
my_list = [8, 10, 6, 2, 4] # list to sort
for i in range(len(my_list) - 1): # we need (5 - 1) comparisons
    if my_list[i] > my_list[i + 1]: # compare adjacent elements
        my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i] # If we end up here, we have to swap the elements.
```

L'algoritmo si può migliorare! Risolviamo questo problema nel modo seguente: introduciamo un'altra variabile, il cui compito è osservare se durante il passaggio è stato effettuato uno scambio o meno; se non c'è scambio, allora l'elenco è già ordinato e non è necessario fare altro. Creiamo una variabile chiamata `swapped` e le assegniamo il valore `False`, per indicare che non ci sono scambi. In caso contrario, viene assegnato il valore `True`

```
my_list = [8, 10, 6, 2, 4] # list to sort
swapped = True # It's a little fake, we need it to enter the while loop.
while swapped:
    swapped = False # no swaps so far
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True # a swap occurred!
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]
print(my_list)
```

### **L'ordinamento a bolle - versione interattiva**

Di seguito si può vedere un programma completo, arricchito da una conversazione con l'utente, che permette di inserire e stampare elementi dall'elenco: L'ordinamento a bolle - versione finale interattiva

```
my_list = []
swapped = True
num = int(input("How many elements do you want to sort: "))

for i in range(num):
    val = float(input("Enter a list element: "))
    my_list.append(val)

while swapped:
    swapped = False
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]

print("\nSorted:")
print(my_list)
```

## L'ordinamento a bolle - versione interattiva

Python, tuttavia, ha i suoi meccanismi di ordinamento. Non è necessario scrivere i propri ordinamenti, perché esiste un numero sufficiente di strumenti pronti all'uso.

Abbiamo spiegato questo sistema di ordinamento perché è importante imparare a elaborare i contenuti di un elenco e per mostrare come può funzionare un vero ordinamento.

Se si vuole che Python ordini l'elenco, si può fare in questo modo:

```
my_list = [8, 10, 6, 2, 4]  
my_list.sort()  
print(my_list)
```

È così semplice.

Il risultato dello snippet è il seguente:

```
[2, 4, 6, 8, 10]
```

Come si può vedere, tutti gli elenchi hanno un metodo chiamato `sort()`, che li ordina il più velocemente possibile. Abbiamo già imparato a conoscere alcuni dei metodi degli elenchi e presto ne impareremo altri.

### **Esercizio 1**

Qual è l'output del seguente snippet?

```
lst = ["D", "F", "A", "Z"]  
lst.sort()
```

```
print(lst)
```

### **Esercizio 2**

Qual è l'output del seguente snippet?

```
a = 3  
b = 1  
c = 2
```

```
lst = [a, c, b]  
lst.sort()
```

```
print(lst)
```

### Esercizio 3

Qual è l'output del seguente snippet?

```
a = "F"  
b = "A"  
c = "C"  
d = " "
```

```
lst = [a, b, c, d]  
lst.reverse()
```

```
print(lst)
```

### Esercizio 4

Si vuole ordinare una lista in ordine decrescente. Scrivi il codice.

## Il ciclo di vita delle liste

Ora vogliamo mostrare un'importante e sorprendente caratteristica delle liste, che le distingue fortemente dalle variabili ordinarie.

Vogliamo che lo memorizzate: potrebbe influenzare i vostri programmi futuri e causare gravi problemi se dimenticato o trascurato.

Date un'occhiata allo snippet nell'editor.

```
list_1 = [1]
list_2 = list_1
list_1[0] = 2
print(list_2)
```

Il programma:

- crea un elenco a un solo elemento, chiamato lista\_1;
- lo assegna a un nuovo elenco chiamato lista\_2;
- cambia l'unico elemento dell'elenco\_1;
- stampa l'elenco\_2.

La parte sorprendente è il fatto che il programma produce l'output [2] e non [1], che sembra essere la soluzione più ovvia.

Quello che succede è che list\_1 e list\_2 sono lo stesso oggetto. Quindi ogni operazione fatta su una lista si rifletterà anche sull'altra (essendo lo stesso elemento, semplicemente identificato da un altro nome)



## Slices

Fortunatamente la soluzione è a portata di mano: si chiama slice.

Una slice è un elemento della sintassi Python che consente di creare una copia nuova di zecca di un elenco, o di parti di esso.

In realtà copia il contenuto dell'elenco, non il nome dell'elenco.

Questo è esattamente ciò di cui avete bisogno. Date un'occhiata allo snippet qui sotto:

```
list_1 = [1]
list_2 = list_1[:]
list_1[0] = 2
print(list_2)
```

Il suo output è [1].

Questa parte poco appariscente del codice, descritta come [:], è in grado di produrre un nuovo elenco.

Una delle forme più generali della slice è la seguente:

```
my_list[inizio:fine]
```

Come si può vedere, assomiglia all'indicizzazione, ma i due punti all'interno fanno una grande differenza.

Una slice di questa forma crea un nuovo elenco (di destinazione), prendendo gli elementi dall'elenco di origine - gli elementi degli indici da inizio a fine - 1.

**Nota:** non a fine ma a fine - 1. Un elemento con indice uguale a fine è il primo elemento che non partecipa alla slice.

## Slices – altri esempi

**# Copia dell'intero elenco.**

```
list_1 = [1]
```

```
lista_2 = lista_1[:]
```

```
lista_1[0] = 2
```

```
print(elenco_2)
```

**# Copia di una parte dell'elenco.**

```
my_list = [10, 8, 6, 4, 2]
```

```
new_list = my_list[1:3]
```

```
print(new_list)
```

## Slices - indici negative

Guardate lo snippet qui sotto:

```
my_list = [10, 8, 6, 4, 2]  
new_list = my_list[-4:-1]  
print(new_list)
```

Ricordate:

```
my_list[inizio:fine]
```

start è l'indice del primo elemento incluso nella slice;

end è l'indice del primo elemento non incluso nella slice.

L'output dello snippet è:

```
[8, 6, 4]
```

Se l'inizio specifica un elemento più lontano di quello descritto dalla fine (dal punto di vista dell'inizio dell'elenco), la slice sarà vuota

```
my_list = [10, 8, 6, 4, 2]  
new_list = my_list[-1:1]  
print(new_list)
```

## Slices

Se si omette l'inizio nella slice, si presume che si voglia ottenere una slice che inizi con l'elemento con indice 0. In altre parole, la fetta di questa forma:

**`my_list[:end]`**

è un equivalente più compatto di:

**`my_list[0:end]`**

Snippet:

```
my_list = [10, 8, 6, 4, 2]  
new_list = my_list[:3]  
print(new_list)
```

il suo output è: [10, 8, 6]

## Slices

Allo stesso modo, se si omette la fine nella slice, si presume che si voglia far terminare la slice all'elemento con l'indice `len(my_list)`.

In altre parole, la fetta di questa forma:

**`my_list[start:]`**

è un equivalente più compatto di:

**`my_list[start:len(my_list)]`**

Snippet:

```
my_list = [10, 8, 6, 4, 2]  
new_list = my_list[3:]  
print(new_list)
```

Il suo output è quindi: `[4, 2]`

## Slices

Come abbiamo già detto, omettendo sia l'inizio che la fine si ottiene una copia dell'intero elenco:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[:]
print(new_list)
```

L'output dello snippet è: [10, 8, 6, 4, 2].

L'istruzione del, descritta in precedenza, è in grado di cancellare più di un elemento di una lista in una volta sola: può cancellare anche delle fette:

```
my_list = [10, 8, 6, 4, 2]
del my_list[1:3]
print(my_list)
```

**Nota:** in questo caso, lo slice non produce alcun nuovo elenco!

L'output dello snippet è: [10, 4, 2].

## Slices

È possibile anche eliminare tutti gli elementi in una volta sola:

```
my_list = [10, 8, 6, 4, 2]  
del my_list[:]  
print(my_list)
```

L'elenco diventa vuoto e l'output è: [].

Rimuovendo lo slice dal codice, il suo significato cambia radicalmente.

Date un'occhiata:

```
my_list = [10, 8, 6, 4, 2]  
del mio_elenco  
print(my_list)
```

L'istruzione del cancellerà l'elenco stesso, non il suo contenuto.

L'invocazione della funzione print() dall'ultima riga del codice causerà quindi un errore di runtime.

## **Gli operatori in e non in**

Python offre due operatori molto potenti, in grado di esaminare l'elenco per verificare se un valore specifico è memorizzato all'interno dell'elenco o meno.

Questi operatori sono:

**elem in my\_list**

**elem not in my\_list**

Il primo di essi (in) verifica se un dato elemento (il suo argomento sinistro) è attualmente memorizzato da qualche parte all'interno dell'elenco (l'argomento destro); in questo caso l'operatore restituisce True.

Il secondo (not in) verifica se un dato elemento (il suo argomento sinistro) è assente in un elenco; in questo caso l'operatore restituisce True.

Guardate il codice:

```
my_list = [0, 3, 12, 8, 2]  
print(5 in my_list)  
print(5 not in my_list)  
print(12 in my_list)
```

Lo snippet mostra entrambi gli operatori in azione. Riuscite a indovinare il risultato? Eseguite il programma per verificare se avete indovinato.



### Liste: alcuni semplici programmi

Ora vogliamo mostrarvi alcuni semplici programmi che utilizzano gli elenchi.

Il primo cerca di trovare il valore maggiore nell'elenco. Guardate il codice

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
```

```
largest = my_list[0]
```

```
for i in range(1, len(my_list)):
```

```
    if my_list[i] > largest:
```

```
        largest = my_list[i]
```

```
print(largest)
```

Il concetto è piuttosto semplice: assumiamo temporaneamente che il primo elemento sia il più grande e verifichiamo l'ipotesi con tutti gli altri elementi dell'elenco.

Il codice produce 17 (come previsto).

## Liste: alcuni semplici programmi

Il codice può essere riscritto per utilizzare la nuova forma di ciclo for introdotta:

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
```

```
largest = my_list[0]
```

```
for i in my_list:
```

```
    if i > largest:
```

```
        largest = i
```

```
print(largest)
```

Il programma precedente esegue un confronto non necessario, quando il primo elemento viene confrontato con se stesso, ma questo non è affatto un problema. Anche il codice produce 17 (niente di strano).

## Liste: alcuni semplici programmi

Se è necessario risparmiare la potenza del computer, è possibile utilizzare una slice:

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
```

```
largest = my_list[0]
```

```
for i in my_list[1:]:
```

```
    if i > largest:
```

```
        largest = i
```

```
print(largest)
```

La domanda è: quale di queste due azioni consuma più risorse del computer: un solo confronto o lo slice di quasi tutti gli elementi di un elenco?

## Liste: alcuni semplici programmi

Ora cerchiamo di trovare la posizione di un dato elemento all'interno di un elenco:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
to_find = 5
```

```
found = False
```

```
for i in range(len(my_list)):
```

```
    found = my_list[i] == to_find
```

```
    if found:
```

```
        break
```

```
if found:
```

```
    print("Element found at index", i)
```

```
else:
```

```
    print("absent")
```

### Nota:

- il valore di destinazione è memorizzato nella variabile to\_find;
- lo stato attuale della ricerca è memorizzato nella variabile found (True/False)
- quando trovato diventa Vero, il ciclo for viene chiuso.

## Liste: alcuni semplici programmi

Supponiamo di aver scelto i seguenti numeri al lotto: 3, 7, 11, 42, 34 e 49.

I numeri estratti sono: 5, 11, 9, 42, 3, 49.

La domanda è: quanti numeri avete indovinato?

Il programma vi darà la risposta:

```
drawn = [5, 11, 9, 42, 3, 49]
```

```
bets = [3, 7, 11, 42, 34, 49]
```

```
hits = 0
```

```
for number in bets:
```

```
    if number in drawn:
```

```
        hits += 1
```

```
print(hits)
```

### Nota:

- l'elenco dei numeri estratti memorizza tutti i numeri estratti;
- l'elenco delle scommesse memorizza le scommesse effettuate;
- la variabile hits conta i risultati ottenuti.

**L'output del programma è: 4.**

## LAB-22

**Tempo stimato:** 10-15 minuti

**Livello di difficoltà:** Facile

### Obiettivi

Familiarizzare lo studente con:

indicizzazione degli elenchi;

utilizzando gli operatori in e non in.

### Scenario

Immaginate un elenco - non molto lungo, non molto complicato, solo un semplice elenco contenente alcuni numeri interi. Alcuni di questi numeri potrebbero essere ripetuti, e questo è l'indizio. Non vogliamo ripetizioni. Vogliamo che vengano eliminate.

Il vostro compito è quello di scrivere un programma che rimuova tutte le ripetizioni di numeri dall'elenco.

L'obiettivo è avere un elenco in cui tutti i numeri non compaiano più di una volta.

Nota: si presuppone che l'elenco dei sorgenti sia codificato all'interno del codice - non è necessario inserirlo dalla tastiera. Naturalmente, è possibile migliorare il codice e aggiungere una parte in grado di effettuare una conversazione con l'utente e ottenere tutti i dati da quest'ultimo.

Suggerimento: vi invitiamo a creare un nuovo elenco come area di lavoro temporanea - non è necessario aggiornare l'elenco in loco.

Non abbiamo fornito dati di prova, perché sarebbe troppo facile. Si può invece utilizzare il nostro scheletro.

```
my_list = [1, 2, 4, 4, 1, 4, 2, 6, 2, 9]
```

```
# Write your code here.
```

```
print("The list with unique elements only:")
```

```
print(my_list)
```

### **Esercizio 1**

Qual è l'output del seguente snippet?

```
list_1 = ["A", "B", "C"]
```

```
list_2 = list_1
```

```
list_3 = list_2
```

```
del list_1[0]
```

```
del list_2[0]
```

```
print(list_3)
```

### **Esercizio 2**

Qual è l'output del seguente snippet?

```
list_1 = ["A", "B", "C"]
```

```
list_2 = list_1
```

```
list_3 = list_2
```

```
del list_1[0]
```

```
del list_2
```

```
print(list_3)
```

### Esercizio 3

Qual è l'output del seguente snippet?

```
list_1 = ["A", "B", "C"]
```

```
list_2 = list_1
```

```
list_3 = list_2
```

```
del list_1[0]
```

```
del list_2[:]
```

```
print(list_3)
```

### Esercizio 4

Qual è l'output del seguente snippet?

```
list_1 = ["A", "B", "C"]
```

```
list_2 = list_1[:]
```

```
list_3 = list_2[:]
```

```
del list_1[0]
```

```
del list_2[0]
```

```
print(list_3)
```



### Esercizio 5

Inserite in o not in al posto di ??? in modo che il codice produca il risultato atteso.

```
my_list = [1, 2, "in", True, "ABC"]
```

```
print(1 ??? my_list) # outputs True
```

```
print("A" ??? my_list) # outputs True
```

```
print(3 ??? my_list) # outputs True
```

```
print(False ??? my_list) # outputs False
```

## **Elenchi in elenchi**

Le liste possono essere composte da scalari (cioè numeri) e da elementi di struttura molto più complessa (avete già visto esempi come stringhe, booleani o altre liste nelle precedenti lezioni di Sintesi delle sezioni). Vediamo più da vicino il caso in cui gli elementi di una lista sono solo liste.

Nella nostra vita troviamo spesso matrici di questo tipo. L'esempio migliore è probabilmente una scacchiera.

Una scacchiera è composta da righe e colonne. Ci sono otto righe e otto colonne. Ogni colonna è contrassegnata dalle lettere da A a H. Ogni riga è contrassegnata da un numero da uno a otto.

La posizione di ogni campo è identificata da coppie lettera-digitale. Così, sappiamo che l'angolo in basso a sinistra della scacchiera (quello con la torre bianca) è A1, mentre l'angolo opposto è H8.

Supponiamo di poter utilizzare i numeri selezionati per rappresentare qualsiasi pezzo degli scacchi. Possiamo anche supporre che ogni riga della scacchiera sia una lista.

Guardate il codice qui sotto:

```
row = []  
for i in range(8):  
    row.append(WHITE_PAWN)
```

Costruisce una lista contenente otto elementi che rappresentano la seconda fila della scacchiera, quella piena di pedoni (si supponga che WHITE\_PAWN sia un simbolo predefinito che rappresenta un pedone bianco).

## Elenchi in elenchi

Lo stesso effetto può essere ottenuto con una list comprehension, la sintassi speciale utilizzata da Python per riempire elenchi enormi.

La list comprehension è in realtà un elenco, ma creato al volo durante l'esecuzione del programma e non descritto staticamente.

Date un'occhiata allo snippet

```
row = [WHITE_PAWN for i in range(8)]
```

La parte di codice posta all'interno delle parentesi specifica:

- i dati da utilizzare per riempire l'elenco (WHITE\_PAWN)
- la clausola che specifica quante volte i dati si trovano all'interno dell'elenco (for i in range(8))

Mostriamo alcuni altri esempi di comprensione delle liste:

Esempio n. 1:

```
squares = [x ** 2 for x in range(10)]
```

Lo snippet produce un elenco di dieci elementi riempito con quadrati di dieci numeri interi a partire da zero (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

## Elenchi in elenchi

Esempio 2:

```
twos = [2 ** i for i in range(8)]
```

Lo snippet crea un array di otto elementi che contiene le prime otto potenze di due (1, 2, 4, 8, 16, 32, 64, 128)

Esempio n. 3:

```
squares = [x ** 2 for x in range(10)]  
print(squares)  
odds = [x for x in squares if x % 2 != 0 ]  
print(odds)
```

Lo snippet crea un elenco con solo gli elementi dispari dell'elenco dei quadrati.

## Elenchi negli elenchi: matrici bidimensionali

Supponiamo anche che un simbolo predefinito chiamato VUOTO indichi un campo vuoto sulla scacchiera.

Quindi, se si vuole creare una lista di liste che rappresenti l'intera scacchiera, si può procedere nel modo seguente:

```
EMPTY=' '  
board = []  
for i in range(8):  
    row = [EMPTY for i in range(8)]  
    board.append(row)  
print(board)
```

### Nota:

- la parte interna del ciclo crea una riga composta da otto elementi (ognuno dei quali è uguale a EMPTY) e la aggiunge all'elenco board;
- la parte esterna lo ripete otto volte;
- in totale, l'elenco board è composto da 64 elementi (tutti uguali a EMPTY)

Questo modello imita perfettamente la scacchiera reale, che è in realtà un elenco di otto elementi, tutti a riga singola. Riassumiamo le nostre osservazioni:

- gli elementi delle righe sono campi, otto per riga;
- gli elementi della scacchiera sono righe, otto per scacchiera.

La variabile board è ora una matrice bidimensionale. Per analogia con i termini algebrici, viene anche chiamata matrice.

## Elenchi negli elenchi: matrici bidimensionali

Poiché le list comprehensions possono essere annidate, possiamo abbreviare la creazione di board nel modo seguente:

```
board = [[EMPTY for i in range(8)] for j in range(8)]
```

L'accesso al campo selezionato della matrice richiede due indici: il primo seleziona la riga; il secondo il numero del campo all'interno della riga, che di fatto è un numero di colonna.

Osservate la scacchiera. Ogni campo contiene una coppia di indici che devono essere forniti per accedere al suo contenuto:

	A	B	C	D	E	F	G	H	
8	[0] [0]	[0] [1]	[0] [2]	[0] [3]	[0] [4]	[0] [5]	[0] [6]	[0] [7]	8
7	[1] [0]	[1] [1]	[1] [2]	[1] [3]	[1] [4]	[1] [5]	[1] [6]	[1] [7]	7
6	[2] [0]	[2] [1]	[2] [2]	[2] [3]	[2] [4]	[2] [5]	[2] [6]	[2] [7]	6
5	[3] [0]	[3] [1]	[3] [2]	[3] [3]	[3] [4]	[3] [5]	[3] [6]	[3] [7]	5
4	[4] [0]	[4] [1]	[4] [2]	[4] [3]	[4] [4]	[4] [5]	[4] [6]	[4] [7]	4
3	[5] [0]	[5] [1]	[5] [2]	[5] [3]	[5] [4]	[5] [5]	[5] [6]	[5] [7]	3
2	[6] [0]	[6] [1]	[6] [2]	[6] [3]	[6] [4]	[6] [5]	[6] [6]	[6] [7]	2
1	[7] [0]	[7] [1]	[7] [2]	[7] [3]	[7] [4]	[7] [5]	[7] [6]	[7] [7]	1
	A	B	C	D	E	F	G	H	

## **Elenchi negli elenchi: matrici bidimensionali**

Osservando la figura qui sopra, mettiamo sulla scacchiera alcuni pezzi degli scacchi. Per prima cosa, aggiungiamo tutte le torri:

**board[0][0] = ROOK**

**board[0][7] = ROOK**

**board[7][0] = ROOK**

**board[7][7] = ROOK**

Se si vuole aggiungere un cavaliere a C4, si procede come segue:

**board[4][2] = KNIGHT**

E ora un pedone in E5:

**board[3][4] = PAWN**

## Elenchi negli elenchi: matrici bidimensionali

Provate il codice seguente

```
EMPTY = "-"
```

```
ROOK = "ROOK"
```

```
board = []
```

```
for i in range(8):
```

```
    row = [EMPTY for i in range(8)]
```

```
    board.append(row)
```

```
board[0][0] = ROOK
```

```
board[0][7] = ROOK
```

```
board[7][0] = ROOK
```

```
board[7][7] = ROOK
```

```
print(board)
```



## Natura multidimensionale delle liste: applicazioni avanzate

Approfondiamo la natura multidimensionale degli elenchi. Per trovare un qualsiasi elemento di un elenco bidimensionale, è necessario utilizzare due *coordinate*:

- una verticale (numero di riga)
- e uno orizzontale (numero di colonna).

Immaginate di sviluppare un software per una stazione meteorologica automatica. Il dispositivo registra la temperatura dell'aria su base oraria e lo fa per tutto il mese. Si ottiene così un totale di  $24 \times 31 = 744$  valori.

Proviamo a progettare una lista in grado di memorizzare tutti questi risultati.

Innanzitutto, è necessario decidere quale tipo di dati sia adeguato per questa applicazione. In questo caso, sarebbe meglio un float, poiché questo termometro è in grado di misurare la temperatura con una precisione di 0,1 °C.

Poi si decide arbitrariamente che le righe registreranno le letture ogni ora (quindi la riga avrà 24 elementi) e ciascuna delle righe sarà assegnata a un giorno del mese (supponiamo che ogni mese abbia 31 giorni, quindi sono necessarie 31 righe). Ecco la coppia di funzioni appropriate (h sta per ora, d per giorno):

```
temps = [[0.0 for h in range(24)] for d in range(31)]
```

L'intera matrice è ora piena di zeri. Si può supporre che venga aggiornata automaticamente utilizzando agenti hardware speciali. È necessario attendere che la matrice venga riempita di misure.

## Natura multidimensionale delle liste: applicazioni avanzate

Ora è il momento di determinare la temperatura media mensile di mezzogiorno. Sommare tutte e 31 le letture registrate a mezzogiorno e dividere la somma per 31. Si può presumere che la temperatura di mezzanotte venga memorizzata per prima. Ecco il codice pertinente:

```
temps = [[0.0 for h in range(24)] for d in range(31)]  
# The matrix is magically updated here.  
total = 0.0  
for day in temps:  
    total += day[11]  
average = total / 31  
print("Average temperature at noon:", average)
```

**Nota:** la variabile giorno utilizzata dal ciclo for non è uno scalare - ogni passaggio attraverso la matrice delle temperature la assegna alle righe successive della matrice; quindi, è un elenco. Deve essere indicizzata con 11 per accedere al valore della temperatura misurata a mezzogiorno.

## Natura multidimensionale delle liste: applicazioni avanzate

Ora trovate la temperatura più alta dell'intero mese - vedi il codice:

```
highest = -100.0
for day in temps:
    for temp in day:
        if temp > highest:
            highest = temp
print("The highest temperature was:", highest)
```

### Nota:

- la variabile giorno itera tutte le righe della matrice temps;
- la variabile temp itera tutte le misurazioni effettuate

Ora contate i giorni in cui la temperatura a mezzogiorno era di almeno 20 °C:

```
hot_days = 0
for day in temps:
    if day[11] > 20.0:
        hot_days += 1
print(hot_days, "days were hot.")
```

## Perché abbiamo bisogno di funzioni?

Finora vi siete imbattuti nelle **funzioni** molte volte, ma la visione dei loro meriti che vi abbiamo fornito è stata piuttosto unilaterale. Le funzioni sono state utilizzate solo come strumenti per rendere la vita più facile e per semplificare compiti lunghi e noiosi.

Quando si vuole che alcuni dati vengano stampati sulla console, si usa `print()`. Quando si vuole leggere il valore di una variabile, si usa `input()`, abbinato a `int()` o `float()`.

Si è anche fatto uso di alcuni **metodi**, che sono in realtà funzioni, ma dichiarate in un modo molto specifico.

Ora imparerete a scrivere le vostre funzioni e a usarle. Scriveremo insieme diverse funzioni, da quelle molto semplici a quelle piuttosto complesse, che richiederanno la vostra attenzione e concentrazione.

Capita spesso che un particolare pezzo di codice venga **ripetuto più volte nel vostro programma**. Viene ripetuto alla lettera o con poche modifiche, che consistono nell'uso di altre variabili nello stesso algoritmo. Capita anche che un programmatore non riesca a resistere alla semplificazione del lavoro e inizi a clonare tali pezzi di codice utilizzando gli appunti e le operazioni di copia-incolla.

Potrebbe risultare molto frustrante quando improvvisamente si scopre che c'era un errore nel codice clonato. Il programmatore dovrà faticare molto per trovare tutti i punti che necessitano di correzioni. C'è anche un alto rischio che le correzioni causino errori.

Possiamo ora definire la prima condizione che può aiutarvi a decidere quando iniziare a scrivere le vostre funzioni: **se un particolare frammento di codice inizia a comparire in più punti, considerate la possibilità di isolarlo sotto forma di funzione** invocata dai punti in cui il codice originale era collocato prima.

## Perché abbiamo bisogno di funzioni?

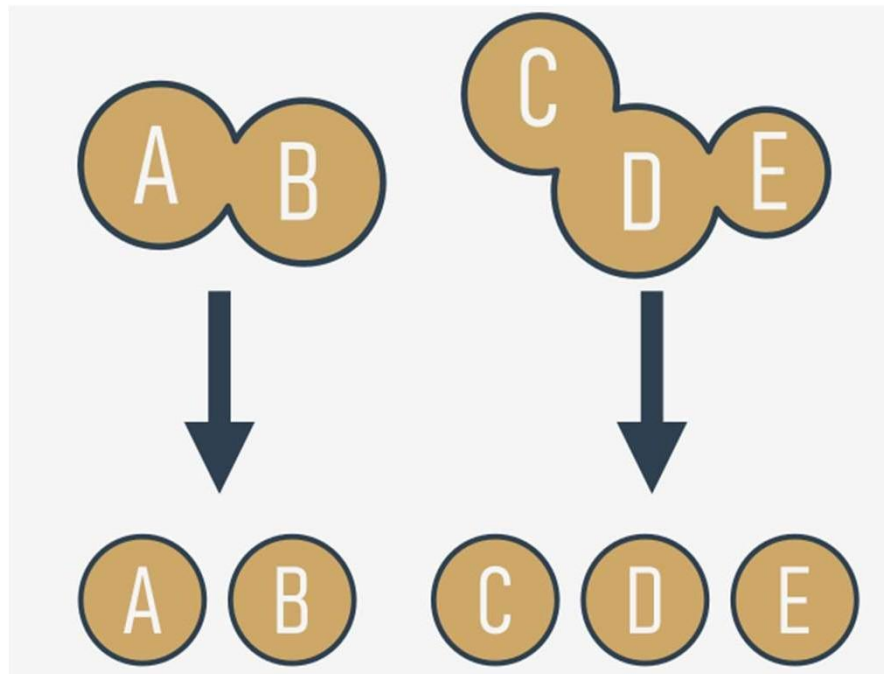
Può accadere che l'algoritmo che si intende implementare sia così complesso che il codice inizia a crescere in modo incontrollato e improvvisamente ci si accorge di non essere più in grado di navigare con facilità.

Si può provare a risolvere il problema commentando ampiamente il codice, ma presto si scopre che questo peggiora drasticamente la situazione: **troppi commenti rendono il codice più grande e più difficile da leggere.**

Alcuni sostengono che una **funzione ben scritta dovrebbe essere vista interamente con un solo sguardo.**

Uno sviluppatore bravo e attento **divide il codice** (o più precisamente: il problema) in pezzi ben isolati e **codifica ciascuno di essi sotto forma di funzione.**

Questo semplifica notevolmente il lavoro del programma, perché ogni pezzo di codice può essere codificato separatamente e testato separatamente. Il processo qui descritto è spesso chiamato **decomposizione.**



## Perché abbiamo bisogno di funzioni?

Possiamo ora enunciare la seconda condizione: **se un pezzo di codice diventa così grande che leggerlo e comprenderlo può causare un problema, prendete in considerazione la possibilità di dividerlo in problemi separati e più piccoli, e di implementare ciascuno di essi sotto forma di una funzione separata.**

Questa scomposizione continua fino a ottenere un insieme di funzioni brevi, facili da capire e da testare.

.

## Decomposizione

Spesso accade che il problema sia così grande e complesso da non poter essere assegnato a un singolo sviluppatore e che un team di sviluppatori debba lavorarci. Il problema deve essere suddiviso tra più sviluppatori in modo da garantire una cooperazione efficiente e senza interruzioni.

Sembra inconcepibile che più di un programmatore scriva lo stesso pezzo di codice nello stesso momento, quindi il lavoro deve essere suddiviso tra tutti i membri del team.

Questo tipo di decomposizione ha uno scopo diverso da quello descritto in precedenza: non si tratta solo di condividere il lavoro, ma anche di **dividere la responsabilità tra molti sviluppatori**.

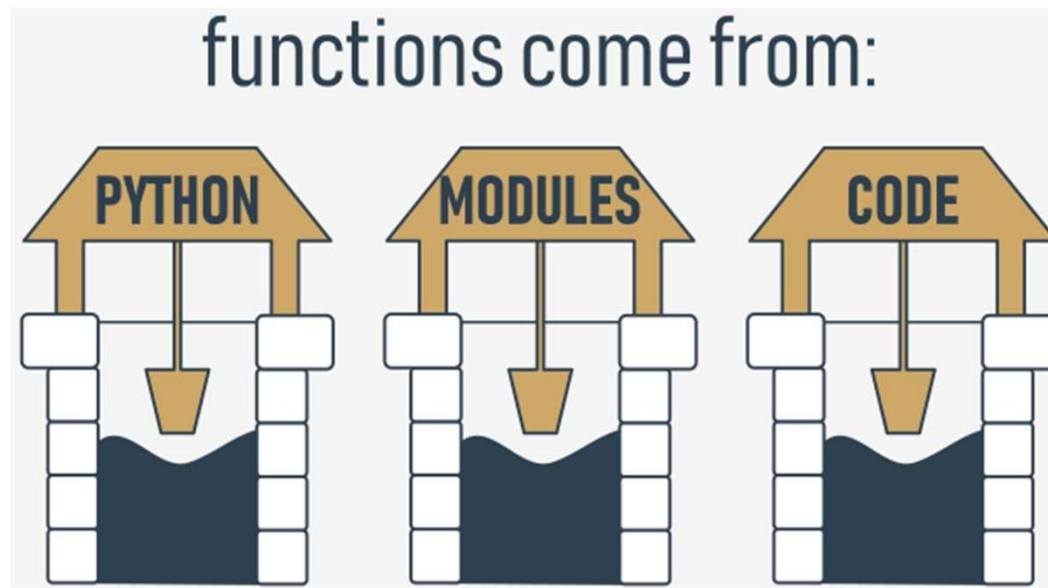
Ognuno di essi scrive un insieme di funzioni chiaramente definite e descritte, che combinate nel modulo (ne parleremo più avanti) daranno il prodotto finale.

Questo ci porta direttamente alla terza condizione: se avete intenzione di dividere il lavoro tra più programmatori, **scomponete il problema** in modo che il prodotto possa essere implementato come un insieme di funzioni scritte separatamente e riunite in moduli diversi.

## Da dove vengono le funzioni?

In generale, le funzioni provengono da almeno tre punti:

- da Python stesso: numerose funzioni (come `print()`) sono parte integrante di Python e sono sempre disponibili senza alcuno sforzo aggiuntivo da parte del programmatore; chiamiamo queste funzioni built-in;
- dai moduli preinstallati di Python - molte funzioni, molto utili, ma usate molto meno spesso di quelle integrate, sono disponibili in una serie di moduli installati insieme a Python; l'uso di queste funzioni richiede alcuni passaggi aggiuntivi da parte del programmatore per renderle pienamente accessibili (ve ne parleremo tra poco);
- direttamente dal vostro codice: potete scrivere le vostre funzioni, inserirle nel vostro codice e usarle liberamente;
- Esiste un'altra possibilità, ma è legata alle classi, quindi per ora la tralasciamo.





## La vostra prima funzione

Date un'occhiata allo snippet

```
print("Enter a value: ")
a = int(input())
print("Enter a value: ")
b = int(input())
print("Enter a value: ")
c = int(input())
```

È piuttosto semplice, ma vogliamo che sia solo un esempio di trasformazione di una parte ripetuta di codice in una funzione.

I messaggi inviati alla console dalla funzione `print()` sono sempre gli stessi. Naturalmente non c'è nulla di male in questo codice, ma provate a immaginare cosa dovrete fare se il vostro capo vi chiedesse di cambiare il messaggio per renderlo più educato, ad esempio iniziandolo con la frase "Per favore".

Sembra che si debba passare un po' di tempo a modificare tutte le occorrenze del messaggio (si potrebbe usare una clipboard, naturalmente, ma non renderebbe la vita molto più facile). È ovvio che si commetterebbero degli errori durante il processo di modifica e che si diventerebbe un po' frustrati (e il proprio capo).

È possibile separare questa parte *ripetibile* del codice, darle un nome e renderla riutilizzabile? Ciò significherebbe che una modifica apportata una volta in un punto verrebbe propagata in tutti i punti in cui viene utilizzata.

Naturalmente, un codice di questo tipo dovrebbe funzionare solo quando viene lanciato esplicitamente.

Sì, è possibile. Le funzioni servono proprio a questo.

## La vostra prima funzione

Come si crea una funzione di questo tipo?

È necessario definirlo. La parola *definire* è significativa in questo caso.

Ecco come si presenta la definizione di funzione

```
def function_name():  
    function_body
```

- Inizia sempre con la parola chiave def (per *definire*)
- dopo def va il nome della funzione (le regole per nominare le funzioni sono esattamente le stesse per nominare le variabili)
- dopo il nome della funzione, c'è un posto per una coppia di parentesi (qui non contengono nulla, ma questo cambierà presto)
- la riga deve terminare con i due punti
- la riga subito dopo def inizia il corpo della funzione - un paio (almeno una) di istruzioni necessariamente annidate, che verranno eseguite ogni volta che la funzione viene invocata; attenzione: la funzione termina dove termina l'annidamento, quindi bisogna fare attenzione.

Siamo pronti a definire la nostra funzione di prompt. La chiameremo messaggio: eccola qui:

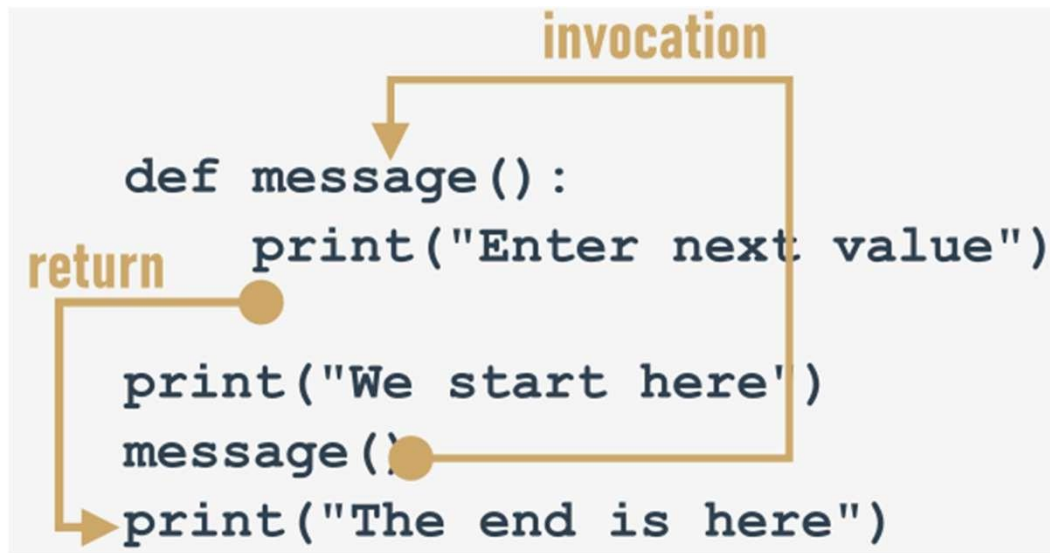
**La vostra prima funzione**

```
def message():  
    print("Enter a value: ")
```

```
print("We start here.")  
message()  
print("We end here.")
```

## Come funzionano le funzioni

Guardate l'immagine qui sotto:



Cerca di mostrare l'intero processo:

- quando si richiama una funzione, Python ricorda il punto in cui è avvenuto e *salta* alla funzione invocata;
- il corpo della funzione viene quindi eseguito;
- raggiungere la fine della funzione costringe Python a tornare al punto direttamente successivo al punto di invocazione.

Ci sono due accorgimenti molto importanti. Ecco il primo:

- Non si deve invocare una funzione che non è nota al momento dell'invocazione.
- Ricordate: Python legge il codice da cima a fondo. Non guarderà avanti per trovare una funzione che avete dimenticato di mettere nel posto giusto ("giusto" significa "prima dell'invocazione").

## Come funzionano le funzioni

Torniamo al nostro esempio principale e utilizziamo la funzione per il lavoro giusto, come in questo caso:

```
def message():  
    print("Enter a value: ")
```

```
message()  
a = int(input())  
message()  
b = int(input())  
message()  
c = int(input())
```

### **Esercizio 1**

La funzione `input()` è un esempio di:

- a) funzione definita dall'utente
- b) funzione integrata

### **Esercizio 2**

Cosa succede quando si cerca di invocare una funzione prima di averla definita? Esempio:

`hi()`

```
def hi():  
    print("hi!")
```

### **Esercizio 3**

Cosa succede quando si esegue il codice sottostante?

```
def hi():  
    print("hi")
```

`hi(5)`

## **Funzioni parametrizzate**

La piena potenza della funzione si rivela quando può essere dotata di un'interfaccia in grado di accettare i dati forniti dall'invocatore. Tali dati possono modificare il comportamento della funzione, rendendola più flessibile e adattabile a condizioni mutevoli.

Un parametro è in realtà una variabile, ma ci sono due fattori importanti che rendono i parametri diversi e speciali:

- I parametri esistono solo all'interno delle funzioni in cui sono stati definiti e l'unico posto in cui il parametro può essere definito è uno spazio tra una coppia di parentesi nell'istruzione `def`;
- L'assegnazione di un valore al parametro avviene al momento dell'invocazione della funzione, specificando l'argomento corrispondente.

**`def function(parametro):`**

**`###`**

Non dimenticare:

- i parametri vivono all'interno delle funzioni (questo è il loro ambiente naturale)
- esistono al di fuori delle funzioni e sono portatori dei valori passati ai parametri corrispondenti.

Esiste una frontiera chiara e inequivocabile tra questi due mondi.

## Funzioni parametrizzate

Arricchiamo la funzione precedente con un solo parametro: lo useremo per mostrare all'utente il numero di un valore richiesto dalla funzione.

Dobbiamo ricostruire la dichiarazione def: ecco come appare ora:

```
def messaggio(numero):
```

```
    ###
```

La definizione specifica che la nostra funzione opera su un solo parametro, chiamato numero. È possibile utilizzarlo come una normale variabile, ma solo all'interno della funzione, non è visibile altrove.

Miglioriamo ora il corpo della funzione:

```
def messaggio(numero):
```

```
    print("Inserisci un numero:", numero)
```

Abbiamo utilizzato il parametro. **Nota:** non abbiamo assegnato al parametro alcun valore.

Un valore per il parametro arriverà quando la funzione sarà invocata.

Ricordate: anche la specificazione di uno o più parametri nella definizione di una funzione è un requisito da soddisfare durante l'invocazione. È necessario fornire tanti argomenti quanti sono i parametri definiti



## **Funzioni parametrizzate**

Modifichiamo la funzione: ora ha due parametri

```
def message(what, number):  
    print("Enter", what, "number", number)
```

```
message("telephone", 11)
```

```
message("price", 5)
```

```
message("number", "number")
```

## Passaggio di parametri posizionali

Una tecnica che assegna l'argomento  $i^{\text{th}}$  (primo, secondo e così via) al parametro  $i^{\text{th}}$  (primo, secondo e così via) della funzione si chiama passaggio di parametri posizionali, mentre gli argomenti passati in questo modo sono chiamati argomenti posizionali.

Lo avete già usato, ma Python può offrire molto di più. Ve ne parliamo ora.

```
def my_function(a, b, c):  
    print(a, b, c)
```

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

**Nota:** il passaggio di parametri posizionali è usato intuitivamente dalle persone in molte occasioni sociali. Ad esempio, è generalmente accettato che quando ci presentiamo menzioniamo il nostro nome (o i nostri nomi) prima del nostro cognome, ad esempio "Mi chiamo John Doe".

Per inciso, gli ungheresi lo fanno in ordine inverso.

Implementiamo questa consuetudine sociale in Python. La seguente funzione si occuperà di introdurre qualcuno:

```
def introduction(first_name, last_name):  
    print("Hello, my name is", first_name, last_name)
```

```
introduction("Luke", "Skywalker")
```

## Passaggio di parametri posizionali

Immaginiamo ora che la stessa funzione venga utilizzata in Ungheria. In questo caso, il codice avrebbe il seguente aspetto:

```
def introduction(first_name, last_name):  
    print("Hello, my name is", first_name, last_name)
```

```
introduction("Skywalker", "Luke")  
introduction("Quick", "Jesse")  
introduction("Kent", "Clark")
```

Il risultato sarà diverso.

È possibile rendere la funzione più indipendente dalla cultura?

## **Passaggio degli argomenti delle parole chiave**

Python offre un'altra convenzione per il passaggio di argomenti, in cui il significato dell'argomento è dettato dal suo nome, non dalla sua posizione: si chiama passaggio di argomenti con parole chiave.

Date un'occhiata allo snippet:

```
def introduction(first_name, last_name):  
    print("Hello, my name is", first_name, last_name)  
  
introduction(first_name = "James", last_name = "Bond")  
introduction(last_name = "Skywalker", first_name = "Luke")
```

Il concetto è chiaro: i valori passati ai parametri sono preceduti dai nomi dei parametri di destinazione, seguiti dal segno =.

La posizione non ha importanza: il valore di ogni argomento conosce la sua destinazione in base al nome utilizzato.

## Mescolare argomenti posizionali e parole chiave

Si possono mischiare entrambi i modi, se si vuole; c'è solo una regola imprescindibile: bisogna mettere gli argomenti posizionali prima di quelli delle parole chiave.

Se riflettete un attimo, indovinerete sicuramente il motivo.

Per mostrarvi come funziona, utilizzeremo la seguente semplice funzione a tre parametri:

```
def adding(a, b, c):
```

```
    print(a, "+", b, "+", c, "=", a + b + c)
```

Il suo scopo è quello di valutare e presentare la somma di tutti i suoi argomenti.

La funzione, quando viene invocata nel modo seguente:

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

Naturalmente, è possibile sostituire tale invocazione con una variante puramente a parole chiave, come questa:

```
adding(c = 1, a = 2, b = 3)
```

## Mescolare argomenti posizionali e parole chiave

Proviamo ora a mescolare entrambi gli stili.

Osservate l'invocazione della funzione qui sotto:

**adding(3, c = 1, b = 2)**

- l'argomento (3) per il parametro a viene passato utilizzando il modo posizionale;
- gli argomenti di c e b sono specificati come parole chiave.

Fate attenzione e fate attenzione agli errori. Se si cerca di passare più di un valore a un argomento, si otterrà solo un errore di runtime.

Guardate l'invocazione qui sotto: sembra che si sia cercato di impostare due volte un valore:

**adding(3, a = 1, b = 2)**

La risposta di Python:

`TypeError: adding() got multiple values for argument 'a'`

## Funzioni parametrizzate: maggiori dettagli

A volte capita che i valori di un particolare parametro siano utilizzati più spesso di altri. Tali argomenti possono avere i loro valori predefiniti (predefiniti) quando i loro argomenti corrispondenti sono stati omessi.

Si dice che il cognome inglese più diffuso sia *Smith*. Proviamo a tenerne conto.

Il valore del parametro predefinito viene impostato utilizzando una sintassi chiara e illustrata:

```
def introduction(first_name, last_name="Smith"):  
    print("Hello, my name is", first_name, last_name)
```

È sufficiente estendere il nome del parametro con il segno =, seguito dal valore predefinito.

Richiamiamo la funzione come di consueto:

```
introduction("James", "Doe")  
introduzione("Henry") #last_name="Smith"  
introduction(first_name="William") #last_name="Smith"
```

**Es. 1**

```
def subtra(a, b):  
    print(a - b)
```

```
subtra(5, 2)  
subtra(2, 5)
```

**Es. 2**

```
def subtra(a, b):  
    print(a - b)
```

```
subtra(a=5, b=2)  
subtra(b=2, a=5)
```

**Es. 3**

```
def subtra(a, b):  
    print(a - b)
```

```
subtra(5, b=2)  
subtra(5, 2)
```



È importante ricordare che gli argomenti posizionali non devono seguire quelli delle parole chiave. Ecco perché se si prova a eseguire il seguente snippet:

```
def subtra(a, b):  
    print(a - b)
```

```
subtra(5, b=2)    # outputs: 3  
subtra(a=5, 2)    # Syntax Error
```

Python non vi permetterà di farlo segnalando un `SyntaxError`.

### **Esercizio 1**

Qual è l'output del seguente snippet?

```
def intro(a="James Bond", b="Bond"):  
    print("My name is", b + ".", a + ".")  
  
intro()
```

### **Esercizio 2**

Qual è l'output del seguente snippet?

```
def intro(a="James Bond", b="Bond"):  
    print("My name is", b + ".", a + ".")  
  
intro(b="Sean Connery")
```

### **Esercizio 3**

Qual è l'output del seguente snippet?

```
def intro(a, b="Bond"):  
    print("My name is", b + ".", a + ".")  
  
intro("Susan")
```

#### Esercizio 4

Qual è l'output del seguente snippet?

```
def add_numbers(a, b=2, c):  
    print(a + b + c)
```

```
add_numbers(a=1, c=3)
```

## Effetti e risultati: l'istruzione return

Tutte le funzioni presentate in precedenza hanno un qualche tipo di effetto: producono del testo e lo inviano alla console.

Naturalmente le funzioni, come i loro fratelli matematici, possono avere dei risultati.

Per far sì che le funzioni restituiscano un valore (ma non solo per questo scopo) si utilizza l'istruzione **return**.

Questa parola fornisce un quadro completo delle sue capacità.

**Nota:** è una parola chiave di Python.

L'istruzione return ha due diverse varianti: consideriamole separatamente.

### return senza espressione

La prima consiste nella parola chiave stessa, senza nulla che la segua.

Quando viene utilizzato all'interno di una funzione, provoca la terminazione immediata dell'esecuzione della funzione e il ritorno istantaneo (da cui il nome) al punto di invocazione.

**Nota:** se una funzione non è destinata a produrre un risultato, l'uso dell'istruzione return non è obbligatorio: verrà eseguita implicitamente alla fine della funzione.

In ogni caso, è possibile utilizzarlo per terminare le attività di una funzione su richiesta, prima che il controllo raggiunga l'ultima riga della funzione.

## Effetti e risultati: l'istruzione return

Consideriamo la seguente funzione:

```
def happy_new_year(wishes = True):  
    print("Three...")  
    print("Two...")  
    print("One...")  
    if not wishes:  
        return  
    print("Happy New Year!")  
happy_new_year()  
happy_new_year(False)
```

La seconda invocazione non stamperà il messaggio **Happy New Year!**

## **Restituire con un'espressione**

La seconda variante di ritorno viene estesa con un'espressione:

```
def function():  
    return espressione
```

Le conseguenze dell'utilizzo sono due:

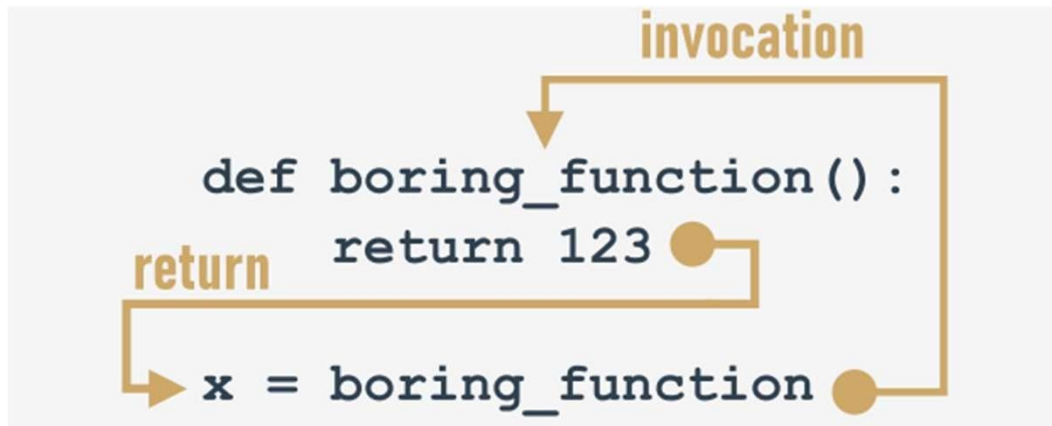
- provoca la terminazione immediata dell'esecuzione della funzione (nulla di nuovo rispetto alla prima variante)
- Inoltre, la funzione valuterà il valore dell'espressione e lo restituirà (da cui il nome ancora una volta) come risultato della funzione.

```
def boring_function():  
    return 123
```

```
x = boring_function()  
print("The boring_function has returned its result. It's:", x)
```

## Restituire con un'espressione

Analizzate la figura sottostante:



L'istruzione `return`, arricchita dall'espressione (in questo caso l'espressione è molto semplice), "trasporta" il valore dell'espressione nel luogo in cui è stata invocata la funzione.

Il risultato può essere utilizzato liberamente, ad esempio per essere assegnato a una variabile.

Può anche essere completamente ignorata e persa senza lasciare traccia.

## Qualche parola su None

Vi presentiamo un valore molto curioso (a dire il vero, un valore nullo) chiamato None.

I suoi dati non rappresentano alcun valore ragionevole, anzi non sono affatto un valore; quindi **non devono partecipare a nessuna espressione**.

Ad esempio, uno snippet come questo:

```
print(None + 2)
```

causerà un errore di runtime, descritto dal seguente messaggio diagnostico:

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

**Nota:** None è una **parola chiave**.

Ci sono solo due tipi di circostanze in cui None può essere usato in modo sicuro:

- quando **lo si assegna a una variabile** (o lo si restituisce come **risultato di una funzione**)
- quando **lo si confronta con una variabile** per diagnosticare il suo stato interno.

Proprio come qui:

```
value = None
```

```
if value is None:
```

```
    print("Sorry, you don't carry any value")
```



## Qualche parola su None

Non dimenticate questo: se una funzione non restituisce un certo valore utilizzando una clausola di espressione `return`, si presume che **restituisca implicitamente** `None`.

Date un'occhiata al codice

```
def strange_function(n):  
    if(n % 2 == 0):  
        return True
```

È ovvio che la funzione `strangeFunction` restituisce `True` quando il suo argomento è pari.

Cosa restituisce altrimenti?

Per verificarlo possiamo utilizzare il seguente codice:

```
print(strange_function(2))  
print(strange_function(1))
```

Questo è ciò che vediamo nella console:

`True`

`None`

## Effetti e risultati: elenchi e funzioni

Ci sono altre due domande a cui bisogna rispondere.

La prima è: **si può inviare un elenco a una funzione come argomento?**

Certo che può! Qualsiasi entità riconosciuta da Python può svolgere il ruolo di argomento di una funzione, anche se bisogna assicurarsi che la funzione sia in grado di gestirla.

Quindi, se si passa un elenco a una funzione, la funzione deve gestirlo come un elenco.

Una funzione come questa qui:

```
def list_sum(lst):
```

```
    s = 0
```

```
    for elem in lst:
```

```
        s += elem
```

```
    return s
```

e invocato in questo modo:

```
print(list_sum([5, 4, 3]))
```

restituirà 12 come risultato, ma bisogna aspettarsi dei problemi se lo si invoca in questo modo rischioso:

```
print(list_sum(5))
```

La risposta di Python sarà inequivocabile:

`TypeError: 'int' object is not iterable`

## Effetti e risultati: elenchi e funzioni

La seconda domanda è: **un elenco può essere il risultato di una funzione?**

Sì, certo! Qualsiasi entità riconoscibile da Python può essere il risultato di una funzione.

Osservate il codice

```
def strange_list_fun(n):  
    strange_list = []  
    for i in range(0, n):  
        strange_list.insert(0, i)  
    return strange_list  
  
print(strange_list_fun(5))
```

L'output del programma sarà simile a questo:

```
[4, 3, 2, 1, 0]
```

Ora è possibile scrivere funzioni con e senza risultati.

Approfondiamo un po' i problemi legati alle variabili nelle funzioni. Questo è essenziale per creare funzioni efficaci e sicure.

## LAB-23

**Tempo stimato:** 10-15 minuti

**Livello di difficoltà:** Facile

### Obiettivi

Familiarizzare lo studente con:  
proiettare e scrivere funzioni parametrizzate;  
utilizzando l'istruzione return;  
verificare le funzioni.

### Scenario

Il vostro compito è scrivere e testare una funzione che prende un argomento (un anno) e restituisce True se l'anno è *bisestile*, o False altrimenti.

Il seme della funzione è già mostrato nel codice scheletrico dell'editor.

Nota: abbiamo preparato anche un breve codice di prova, che potete utilizzare per testare la vostra funzione.

Il codice utilizza due elenchi: uno con i dati del test e l'altro contenente i risultati attesi. Il codice indica se i risultati non sono validi.

## LAB-23

```
def is_year_leap(year):
```

```
#
```

```
# Write your code here.
```

```
#
```

```
test_data = [1900, 2000, 2016, 1987]
```

```
test_results = [False, True, True, False]
```

```
for i in range(len(test_data)):
```

```
    yr = test_data[i]
```

```
    print(yr,"->",end="")
```

```
    result = is_year_leap(yr)
```

```
    if result == test_results[i]:
```

```
        print("OK")
```

```
    else:
```

```
        print("Failed")
```

## **LAB-24**

**Tempo stimato:** 15-20 minuti

**Livello di difficoltà:** Medio

### **Obiettivi**

Familiarizzare lo studente con:  
proiettare e scrivere funzioni parametrizzate;  
utilizzando l'istruzione return;  
utilizzando le funzioni proprie dello studente.

### **Scenario**

Il vostro compito è scrivere e testare una funzione che prenda due argomenti (un anno e un mese) e restituisca il numero di giorni per la coppia mese/anno data (mentre solo febbraio è sensibile al valore dell'anno, la vostra funzione dovrebbe essere universale).

La parte iniziale della funzione è pronta. Ora bisogna convincere la funzione a restituire None se i suoi argomenti non hanno senso.

Naturalmente, è possibile (e opportuno) utilizzare la funzione scritta e testata in precedenza (LAB-23). Può essere molto utile. Vi invitiamo a utilizzare un elenco con le lunghezze dei mesi. Potete crearla all'interno della funzione: questo trucco accorcerà notevolmente il codice.

Abbiamo preparato un codice di test. Espandetelo per includere altri casi di test.

## LAB-24

```
def is_year_leap(year):
```

```
#
```

```
# Your code from LAB-23.
```

```
def days_in_month(year, month):
```

```
#
```

```
# Write your new code here.
```

```
test_years = [1900, 2000, 2016, 1987]
```

```
test_months = [2, 2, 1, 11]
```

```
test_results = [28, 29, 31, 30]
```

```
for i in range(len(test_years)):
```

```
    yr = test_years[i]
```

```
    mo = test_months[i]
```

```
    print(yr, mo, "->", end="")
```

```
    result = days_in_month(yr, mo)
```

```
    if result == test_results[i]:
```

```
        print("OK")
```

```
    else:
```

```
        print("Failed")
```

## **LAB-25**

**Tempo stimato:**20-30 minuti

**Livello di difficoltà:** Medio

### **Prerequisiti**

LAB-23 e LAB-24

### **Obiettivi**

Familiarizzare lo studente con:

proiettare e scrivere funzioni parametrizzate;

utilizzando l'istruzione return;

costruire un insieme di funzioni di utilità;

utilizzando le funzioni proprie dello studente.

### **Scenario**

Il vostro compito è scrivere e testare una funzione che prenda tre argomenti (un anno, un mese e un giorno del mese) e restituisca il giorno dell'anno corrispondente, oppure restituisca Nessuno se uno degli argomenti non è valido.

Utilizzare le funzioni scritte e testate in precedenza. Aggiungere al codice alcuni casi di test. Questo test è solo un inizio.



## LAB-25

```
def is_year_leap(year):  
    #
```

```
def days_in_month(year, month):  
    #
```

```
def day_of_year(year, month, day):  
    #
```

```
print(day_of_year(2000, 12, 31))
```

## LAB-26

**Tempo stimato:** 15-20 minuti

**Livello di difficoltà:** Medio

### Obiettivi

familiarizzare lo studente con nozioni e algoritmi classici;  
migliorare le competenze dello studente nella definizione e nell'uso delle funzioni.

### Scenario

*Un numero naturale è **primo** se è maggiore di 1 e non ha divisori diversi da 1 e da se stesso.*

Complicato? Niente affatto. Per esempio, 8 non è un numero primo, perché si può dividere per 2 e 4 (non si possono usare divisori uguali a 1 e 8, perché la definizione lo vieta).

D'altra parte, 7 è un numero primo, poiché non è possibile trovare alcun divisore legale per esso.

Il vostro compito è scrivere una funzione che verifichi se un numero è primo o no.

La funzione `is_prime`

- prende un parametro (il valore da controllare)
- restituisce `True` se l'argomento è un numero primo e `False` altrimenti.

Suggerimento: provate a dividere l'argomento per tutti i valori successivi (a partire da 2) e controllate il resto: se è zero, il numero non può essere un primo; riflettete attentamente su quando interrompere il processo.

Se avete bisogno di conoscere la radice quadrata di un valore qualsiasi, potete utilizzare l'operatore `**`. Ricordate: la radice quadrata di  $x$  è uguale a  $x^{0.5}$

## LAB-26

Codice:

```
def is_prime(num):  
    #  
    # Write your code here.  
    #  
  
for i in range(1, 20):  
    if is_prime(i + 1):  
        print(i + 1, end=" ")  
  
print()
```

**Risultato previsto**

2 3 5 7 11 13 17 19

## LAB-27

**Tempo stimato:** 10-15 minuti

**Livello di difficoltà:** Facile

### **Obiettivi**

migliorare le competenze dello studente nella definizione, nell'uso e nella verifica delle funzioni.

### **Scenario**

Il consumo di carburante di un'auto può essere espresso in molti modi diversi. Ad esempio, in Europa viene indicato come quantità di carburante consumato per 100 chilometri.

Negli Stati Uniti, viene indicato come il numero di miglia percorse da un'automobile utilizzando un gallone di carburante.

Il vostro compito è quello di scrivere una coppia di funzioni che convertano l/100km in mpg e viceversa.

Le funzioni:

- sono denominati rispettivamente `liters_100km_to_miles_gallon` e `miles_gallon_to_liters_100km` respectively
- prendono un solo argomento (il valore corrispondente ai loro nomi)

Completare il codice seguente

## LAB-27

```
def liters_100km_to_miles_gallon(liters):
```

```
#
```

```
# Write your code here.
```

```
#
```

```
def miles_gallon_to_liters_100km(miles):
```

```
#
```

```
# Write your code here
```

```
#
```

```
print(liters_100km_to_miles_gallon(3.9))
```

```
print(liters_100km_to_miles_gallon(7.5))
```

```
print(liters_100km_to_miles_gallon(10.))
```

```
print(miles_gallon_to_liters_100km(60.3))
```

```
print(miles_gallon_to_liters_100km(31.4))
```

```
print(miles_gallon_to_liters_100km(23.5))
```

## **LAB-27**

Eseguite il vostro codice e verificate se il risultato è uguale al nostro.  
Ecco alcune informazioni per aiutarvi:

1 miglio americano = 1609,344 metri;  
1 gallone americano = 3,785411784 litri.

### **Risultato previsto**

60.31143162393162  
31.36194444444444  
23.52145833333333  
3.9007393587617467  
7.490910297239916  
10.009131205673757

### **Esercizio 1**

Qual è l'output del seguente snippet?

```
def hi():  
    return  
    print("Hi!")
```

```
hi()
```

### **Esercizio 2**

Qual è l'output del seguente snippet?

```
def is_int(data):  
    if type(data) == int:  
        return True  
    elif type(data) == float:  
        return False
```

```
print(is_int(5))  
print(is_int(5.0))  
print(is_int("5"))
```

### Esercizio 3

Qual è l'output del seguente snippet?

```
def even_num_lst(ran):  
    lst = []  
    for num in range(ran):  
        if num % 2 == 0:  
            lst.append(num)  
    return lst  
  
print(even_num_lst(11))
```

### Esercizio 4

Qual è l'output del seguente snippet?

```
def list_updater(lst):  
    upd_list = []  
    for elem in lst:  
        elem **= 2  
        upd_list.append(elem)  
    return upd_list  
  
foo = [1, 2, 3, 4, 5]  
print(list_updater(foo))
```



## Funzioni e ambiti di applicazione

Cominciamo con una definizione:

L'**ambito di un nome** (ad esempio, il nome di una variabile) è la parte di codice in cui il nome è correttamente riconoscibile.

Ad esempio, l'ambito del parametro di una funzione è la funzione stessa. Il parametro è inaccessibile al di fuori della funzione.

Verifichiamo. Guardate il codice

```
def scope_test():
```

```
    x = 123
```

```
scope_test()
```

```
print(x)
```

Cosa succede quando lo si esegue?

Il programma non funziona quando viene eseguito. Il messaggio di errore recita:

NameError: name 'x' is not defined

Questo è prevedibile.

Faremo alcuni esperimenti per mostrarvi come Python costruisce gli ambiti e come potete usare le sue abitudini a vostro vantaggio.

## Funzioni e ambiti di applicazione

Iniziamo controllando se una variabile creata al di fuori di una funzione è visibile o meno all'interno delle funzioni. In altre parole, il nome di una variabile si propaga nel corpo di una funzione?

Guardate il codice

```
def my_function():  
    print("Do I know that variable?", var)
```

```
var = 1  
my_function()  
print(var)
```

La nostra cavia è lì.

Il risultato del test è positivo: il codice emette:

```
Do I know that variable? 1  
1
```

La risposta è: **una variabile esistente al di fuori di una funzione ha un ambito all'interno del corpo della funzione stessa.**

Questa regola ha un'eccezione molto importante. Cerchiamo di trovarla.

## Funzioni e ambiti di applicazione

Apportiamo una piccola modifica al codice:

**Let's make a small change to the code:**

```
def my_function():  
    var = 2  
    print("Do I know that variable?", var)
```

```
var = 1  
my_function()  
print(var)
```

Anche il risultato è cambiato: il codice produce ora un output leggermente diverso:

Do I know that variable? 2

1

Che cosa è successo?

- la variabile `var` creata all'interno della funzione non è la stessa di quella definita al di fuori di essa - sembra che ci siano due variabili diverse con lo stesso nome;
- Inoltre, la variabile della funzione fa da ombra alla variabile proveniente dal mondo esterno.

## Funzioni e ambiti di applicazione

Possiamo rendere la regola precedente più precisa e adeguata:

**Una variabile esistente al di fuori di una funzione ha un ambito all'interno dei corpi delle funzioni, escluse quelle che definiscono una variabile con lo stesso nome.**

Significa anche che l'**ambito di una variabile esistente al di fuori di una funzione è supportato solo quando si ottiene il suo valore** (lettura). L'assegnazione di un valore forza la creazione di una variabile propria della funzione.

## Funzioni e ambiti: la parola chiave globale

Se tutto va bene, dovrete essere arrivati alla seguente domanda: questo significa che una funzione non è in grado di modificare una variabile definita al suo esterno? Questo creerebbe molti disagi.

Fortunatamente, la risposta è *no*.

Esiste un metodo speciale di Python che permette di **estendere l'ambito di una variabile in modo da includere i corpi delle funzioni** (anche se non si vuole solo leggere i valori, ma anche modificarli).

Questo effetto è causato da una parola chiave chiamata `global`:

**`global name`**

**`global name1, name2, ...`**

L'uso di questa parola chiave all'interno di una funzione con il nome (o i nomi separati da virgole) di una o più variabili, obbliga Python a non creare una nuova variabile all'interno della funzione; verrà invece utilizzata quella accessibile dall'esterno.

In altre parole, questo nome diventa globale (ha un **ambito globale** e non importa se è oggetto di lettura o di assegnazione).

## Funzioni e ambiti: la parola chiave globale

Guardate il codice

```
def my_function():  
    global var  
    var = 2  
    print("Do I know that variable?", var)
```

```
var = 1  
my_function()  
print(var)
```

Abbiamo aggiunto la funzione globale.  
Il codice viene ora visualizzato:

```
Do I know that variable? 2  
2
```

Questa dovrebbe essere una prova sufficiente per dimostrare che la parola chiave globale fa ciò che promette.

## **Come la funzione interagisce con i suoi argomenti**

Scopriamo ora come la funzione interagisce con i suoi argomenti.

Il codice

```
def my_function(n):  
    print("I got", n)  
    n += 1  
    print("I have", n)
```

```
var = 1  
my_function(var)  
print(var)
```

dovrebbe insegnare qualcosa. Come si può vedere, la funzione cambia il valore del suo parametro. La modifica ha effetto sull'argomento?

Eseguire il programma e verificare.

L'output del codice è:

```
I got 1  
I have 2  
1
```

### **Come la funzione interagisce con i suoi argomenti**

La conclusione è ovvia: la **modifica del valore del parametro non si propaga all'esterno della funzione** (in ogni caso, non quando la variabile è uno scalare, come nell'esempio).

Ciò significa anche che una funzione riceve **il valore** dell'**argomento**, non l'argomento stesso. Questo vale per gli scalari.

Vale la pena di verificare come funziona con gli elenchi (ricordate le peculiarità dell'assegnazione di fette di elenchi rispetto all'assegnazione di elenchi nel loro complesso?)



## **Come la funzione interagisce con i suoi argomenti**

L'esempio seguente chiarisce il problema:

```
def my_function(my_list_1):  
    print("Print #1:", my_list_1)  
    print("Print #2:", my_list_2)  
    my_list_1 = [0, 1]  
    print("Print #3:", my_list_1)  
    print("Print #4:", my_list_2)
```

```
my_list_2 = [2, 3]  
my_function(my_list_2)  
print("Print #5:", my_list_2)
```

L'output del codice è:

```
Print #1: [2, 3]  
Print #2: [2, 3]  
Print #3: [0, 1]  
Print #4: [2, 3]  
Print #5: [2, 3]
```

Sembra che la prima regola sia ancora valida.

### **Come la funzione interagisce con i suoi argomenti**

Infine, è possibile notare la differenza nell'esempio seguente:

```
def my_function(my_list_1):  
    print("Print #1:", my_list_1)  
    print("Print #2:", my_list_2)  
    del my_list_1[0] # Pay attention to this line.  
    print("Print #3:", my_list_1)  
    print("Print #4:", my_list_2)
```

```
my_list_2 = [2, 3]  
my_function(my_list_2)  
print("Print #5:", my_list_2)
```

Non cambiamo il valore del parametro `my_list_1` (sappiamo già che non influirà sull'argomento), ma modifichiamo l'elenco da esso identificato.

Il risultato potrebbe essere sorprendente. Eseguite il codice e verificate:

```
Print #1: [2, 3]  
Print #2: [2, 3]  
Print #3: [3]  
Print #4: [3]  
Print #5: [3]
```

## **Come la funzione interagisce con i suoi argomenti**

Perchè?

Proviamo a spagarlo:

- se l'argomento è un elenco, la modifica del valore del parametro corrispondente non influisce sull'elenco (ricordate: le variabili contenenti elenchi sono memorizzate in modo diverso dagli scalari),
- ma se si modifica un elenco identificato dal parametro (attenzione: l'elenco, non il parametro!), l'elenco rifletterà la modifica.

### Esercizio 1

Cosa succede quando si prova a eseguire il codice seguente?

```
def message():
```

```
    alt = 1
```

```
    print("Hello, World!")
```

```
print(alt)
```

### Esercizio 2

Qual è l'output del seguente snippet?

```
a = 1
```

```
def fun():
```

```
    a = 2
```

```
    print(a)
```

```
fun()
```

```
print(a)
```

### Esercizio 3

Qual è l'output del seguente snippet?

```
a = 1
def fun():
    global a
    a = 2
    print(a)
```

```
fun()
a = 3
print(a)
```

### Esercizio 4

Qual è l'output del seguente snippet?

```
a = 1
def fun():
    global a
    a = 2
    print(a)
```

```
a = 3
fun()
print(a)
```

### Alcune funzioni semplici: valutazione del BMI

Cominciamo con una funzione per valutare l'indice di massa corporea (BMI).

$$\text{BMI} = \frac{\text{(weight in kilograms)} \text{ 

Come si può vedere, la formula ottiene due valori:$$

- peso (originariamente in chilogrammi)
- altezza (originariamente in metri)

Sembra che questa nuova funzione avrà **due parametri**. Il suo nome sarà bmi, ma se preferite un altro nome, usatelo.

Codifichiamo la funzione:

```
def bmi(weight, height):  
    return weight / height ** 2  
print(bmi(52.5, 1.65))
```

### **Alcune funzioni semplici: valutazione del BMI**

Il risultato prodotto dall'invocazione di esempio è il seguente:

19.283746556473833

La funzione soddisfa le nostre aspettative, ma è un po' semplice: presuppone che i valori di entrambi i parametri siano sempre significativi. Vale sicuramente la pena di verificare se sono attendibili.

Controlliamo entrambi e riportiamo None se uno dei due sembra sospetto.

## Alcune funzioni semplici: valutazione del BMI

Osservate il codice

```
def bmi(weight, height):  
    if height < 1.0 or height > 2.5 or \  
    weight < 20 or weight > 200:  
        return None  
    return weight / height ** 2  
print(bmi(352.5, 1.65))
```

Ci sono due cose a cui dobbiamo prestare attenzione.

In primo luogo, l'invocazione del test assicura che la **funzione restituisca un valore** correttamente: l'output è:

None

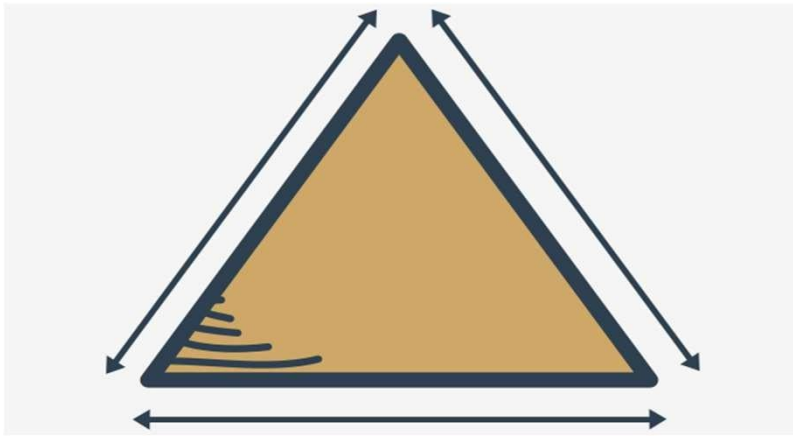
In secondo luogo, osservate il modo in cui viene utilizzato il simbolo **backslash** (\). Se lo si usa nel codice Python e si termina una riga con esso, si indica a Python di continuare la riga di codice nella riga successiva.

Può essere particolarmente utile quando si ha a che fare con lunghe righe di codice e si desidera migliorare la leggibilità del codice.



### Alcune funzioni semplici: continua

Giochiamo ora con i triangoli. Inizieremo con una funzione per verificare se tre lati di lunghezza data possono formare un triangolo.



A scuola sappiamo che *la somma di due lati arbitrari deve essere più lunga del terzo lato*.

Non sarà una sfida difficile. La funzione avrà **tre parametri**, uno per ogni lato.

Restituirà True se i lati possono formare un triangolo e False altrimenti. In questo caso, `is_a_triangolo` è un buon nome per questa funzione.

### **Alcune funzioni semplici: continua**

Guardare il codice

```
def is_a_triangle(a, b, c):
```

```
    if a + b <= c:
```

```
        return False
```

```
    if b + c <= a:
```

```
        return False
```

```
    if c + a <= b:
```

```
        return False
```

```
    return True
```

```
print(is_a_triangle(1, 1, 1))
```

```
print(is_a_triangle(1, 1, 3))
```

Eseguire il programma.

Sembra che funzioni bene: questi sono i risultati:

Vero

Falso

### **Alcune funzioni semplici: continua**

Possiamo renderlo più compatto? Mi sembra un po' troppo lungo.

Questa è una versione più compatta:

```
def is_a_triangle(a, b, c):  
    if a + b <= c or b + c <= a or c + a <= b:  
        return False  
    return True
```

```
print(is_a_triangle(1, 1, 1))  
print(is_a_triangle(1, 1, 3))
```

Possiamo compattarlo ancora di più?

Sì, possiamo - guardate:

```
def is_a_triangle(a, b, c):  
    return a + b > c and b + c > a and c + a > b
```

Abbiamo negato la condizione (invertendo gli operatori relazionali e sostituendo gli or con gli and, ottenendo un'**espressione universale per testare i triangoli**).