

# **Python**

## **GIORNO 2 - Collections, indexing, costrutti logici e loops**

## Agenda:

- ❑ Collections
- ❑ Costrutti logici
- ❑ Loop



## Perché?

- ❑ Come data analyst vi troverete spesso a lavorare con **raggruppamenti di elementi**: è importante quindi sapere quali sono i raggruppamenti base, cosa li caratterizza e quali operazioni si possono fare su di essi.
- ❑ Imporre delle **condizioni logiche** ai nostri algoritmi, e permettere di **ripetere delle operazioni** in sequenza su più elementi, sono operazioni fondamentali per gestire i dati.

## Alla fine di questa lezione saprete:

- ❑ Cosa sono **tuple, set e dizionari**, cosa li caratterizza, come scegliere quello più adatto alle diverse situazioni e come manipolarli
- ❑ Utilizzare il costrutto logico **if-elif-else** per gestire casi condizionali nel vostro codice
- ❑ Utilizzare il **ciclo while** e il **ciclo for** per gestire i casi in cui vogliamo ripetere le stesse operazioni più volte.

## Collections

**Le collections sono strutture dati in un linguaggio di programmazione che consentono di organizzare e manipolare gruppi di elementi in modo efficiente.**

Le principali collections disponibili in Python sono:

- ☐ Liste
- ☐ Tuple
- ☐ Set
- ☐ Dizionari

## Tuple

**Una tupla è una collezione ordinata e immutabile di elementi.**

Per definire una `tuple` utilizziamo le parentesi tonde e separiamo i singoli elementi con una virgola,

per esempio:

```
1 my_tuple = ("tuple", 1, 2, True)
2 print(my_tuple)

('tuple', 1, 2, True)
```

Oppure, se dopo l'operatore di assegnazione `=`, mettiamo un elenco di elementi separati da virgola ma non incluso in alcun tipo di parentesi, Python gli assegnerà di default il datatype `tuple`.

## Tuple — indexing / slicing

Essendo le tuple delle data structure **ordinate**, possiamo utilizzare indexing e slicing esattamente come per le liste,

per esempio:

```
1 my_tuple = ("tuple", 1, 2, True, 12.46, ["una", "lista", "in", "una_tupla"])
2 print(my_tuple[4])
3 print(my_tuple[0:3])

12.46
('tuple', 1, 2)
```

Dato che le tuple sono **immutabili**, non posso modificare gli elementi al loro interno. Se ci provo Python mi restituirà un errore:

```
1 my_tuple = ("tuple", 1, 2, True, 12.46, ["una", "lista", "in", "una_tupla"])
2 my_tuple[4] = "modifico"

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-27-f642aaed915a> in <cell line: 2>()
      1 my_tuple = ("tuple", 1, 2, True, 12.46, ["una", "lista", "in", "una_tupla"])
----> 2 my_tuple[4] = "modifico"

TypeError: 'tuple' object does not support item assignment
```

## Tuple — metodi

Dato che le tuple sono immutabili non hanno metodi di manipolazione: una volta definite non possono essere modificate in alcun modo. Possiamo utilizzare però:

- ❑ `tuple.index(elemento[, start[, stop]])` : Restituisce l'indice della prima occorrenza dell'elemento
- ❑ `tuple.count(elemento)` : Restituisce il numero di occorrenze dell'elemento



## Tuple — metodi

Esempi:

```
quantita_ordini = (2, 3, 1, 1, 4, 3, 7, 3, 9, 2, 3, 5)
```

```
print(quantita_ordini)
```

```
>>> (2, 3, 1, 1, 4, 3, 7, 3, 9, 2, 3, 5)
```

```
print( quantita_ordini.count(3) )
```

```
>>> 4
```

```
orari = 10, 14, 18, 20, 23
```

```
print(orari)
```

```
>>> (10, 14, 18, 20, 23)
```

```
print( orari.index(18) )
```

```
>>> 2
```

```
print( orari[2] )
```

```
>>> 18
```

## Set

**Un set è una collezione non ordinata di elementi unici.**

Per definire un set utilizziamo le parentesi graffe e separiamo i singoli elementi con una virgola,

per esempio:

```
1 my_set = {"un", 1, "set"}
2 print(my_set)

{1, 'set', 'un'}
```

Vediamo che è una collezione **non ordinata** anche dall'output della funzione `print()`: gli elementi non sono nello stesso ordine in cui li abbiamo inseriti. Essendo una struttura non ordinata, non supporta l'indexing.

## Set

La caratteristica fondamentale dei set è l'**unicità degli elementi** che lo compongono,

per esempio:

```
1 my_set = {1, 1, 2, 3, 4, 4, 5, 2}
2 print(my_set)

{1, 2, 3, 4, 5}
```

Vengono esclusi in maniera automatica tutti gli elementi duplicati, di cui viene conservata solo un'occorrenza.

## Set — metodi

I set hanno dei metodi “basici” per essere manipolati, ad esempio:

- ❑ `set.add(elemento)` : Aggiunge un elemento al set. Se l'elemento esiste già, non viene aggiunto nuovamente
- ❑ `set.remove(elemento)` : Rimuove un elemento dal set. Solleva un'eccezione `KeyError` se l'elemento non esiste nel set
- ❑ `set.discard(elemento)` : Rimuove un elemento dal set, se presente. Non solleva un'eccezione se l'elemento non esiste nel set
- ❑ `set.update(iterabile)` : Aggiunge gli elementi dell'iterabile al set.

## Set — metodi

I set hanno anche una serie di metodi “insiemistici”:

- ❑ `set.union(iterabile)` : Restituisce un nuovo set che contiene tutti gli elementi del set originale e dell'iterabile.
- ❑ `set.intersection(iterabile)` : Restituisce un nuovo set che contiene solo gli elementi comuni tra il set originale e l'iterabile.
- ❑ `set.difference(iterabile)` : Restituisce un nuovo set che contiene gli elementi presenti nel set originale ma non nell'iterabile.
- ❑ `set.symmetric_difference(iterabile)` : Restituisce un nuovo set che contiene gli elementi presenti in uno dei due set, ma non in entrambi.
- ❑ `set.issubset(iterabile)` : Restituisce True se il set è un sottoinsieme dell'iterabile.
- ❑ `set.issuperset(iterabile)` : Restituisce True se il set contiene tutti gli elementi dell'iterabile.

## Set — metodi

```
citta_italiane = {"Roma", "Napoli", "Firenze", "Milano", "Palermo"}

citta_italiane.add("Catania")
print(citta_italiane)
>>> {"Roma", "Napoli", "Firenze", "Milano", "Palermo", "Catania"}

citta_italiane.remove("Firenze")
print(citta_italiane)
>>> {"Roma", "Napoli", "Firenze", "Milano", "Palermo", "Catania"}

citta_italiane.update(["Lucca", "Pavia", "Bari"])
print(citta_italiane)
>>> {"Roma", "Napoli", "Firenze", "Milano", "Palermo", "Catania", "Lucca",
     "Pavia", "Bari"}
```

## Set — metodi

```
citta_italiane = {"Roma", "Napoli", "Firenze", "Milano", "Palermo"}
grandi_citta = {"Parigi", "Roma", "Berlino", "Madrid", "Milano"}

citta_italiane.union(grandi_citta)
>>> {"Roma", "Napoli", "Firenze", "Parigi", "Berlino", "Milano", "Palermo",
"Madrid"}

citta_italiane.intersection(grandi_citta)
>>> {"Roma", "Milano"}

grandi_citta.difference(citta_italiane)
>>> {"Parigi", "Berlino", "Madrid"}

grandi_citta.symmetric_difference(citta_italiane)
>>> {"Napoli", "Firenze", "Palermo", "Parigi", "Berlino", "Madrid"}
```

## Set — metodi

```
# Set di utenti registrati e utenti attivi
registered_users = {"Anna", "Luca", "Marco", "Sara"}
active_users = {"Marco", "Sara"}

# Gli utenti attivi sono un sottoinsieme di quelli registrati?
print(active_users.issubset(registered_users))
>>> True

# Gli utenti registrati sono un soprainsieme degli attivi?
print(registered_users.issuperset(active_users))
>>> True

# Ci sono utenti registrati che non sono attivi?
inactive_users = registered_users.difference(active_users)
print(inactive_users)
>>> {'Anna', 'Luca'}
```



## Commenti

Per aiutarci nella scrittura del codice possiamo aggiungere dei **commenti**, ovvero dei testi che verranno ignorati da Python — servono infatti a noi per descrivere quello che stiamo facendo.

I commenti su una sola riga vengono preceduti dal cancelletto #:

- possono essere inseriti su una riga da soli, oppure possono seguire un comando

I commenti multiriga si creano con tre apici doppi `"""` o singoli `'` (come le stringhe multiriga)

```
# Commento su una riga

somma = 5 + 20  # Commento dopo un'istruzione

"""
Commento su più righe, ad esempio:
In questa sezione si analizzano i dati dell'anno precedente
"""
```

## Dizionari

**Un dizionario è una struttura dati che associa coppie chiave-valore. È una collezione mutabile e non ordinata di elementi.**

Per definire un dizionario utilizziamo le parentesi graffe e definiamo le coppie chiave-valore utilizzando i ":" e separandole dalle altre copie con una virgola,

per esempio:

```
1 my_dict = {"prima_chiave": "primo_valore",  
2           2: "secondo_valore",  
3           "terza_chiave": 3}  
4 print(my_dict)  
  
{'prima_chiave': 'primo_valore', 2: 'secondo_valore', 'terza_chiave': 3}
```

Le **chiavi** in un dizionario devono essere **univoche e immutabili** (come stringhe, tuple o numeri), mentre i **valori** possono essere di **qualsiasi tipo di dato**, inclusi altri dizionari.

## Dizionari — indexing

I dizionari sono delle collections **non ordinate**, non supportano quindi l'indexing numerico e lo slicing come fanno le liste e le tuple.

Possiamo però accedere ad un singolo elemento indicizzandolo con la sua chiave,

per esempio:

```
1 my_dict = {"nome": "Mario",  
2           "cognome": "Rossi",  
3           "età": 45}  
4 print(my_dict["cognome"])
```

Rossi

## Dizionari — indexing

Seguendo la stessa logica, posso aggiungere una copia chiave-valore ad un dizionario **istanziando una nuova chiave** ed assegnandole il suo valore,

per esempio:

```
1 my_dict = {"nome": "Mario",  
2           "cognome": "Rossi",  
3           "età": 45}  
4 print(my_dict)  
5 my_dict["occupazione"] = "Data Analyst"  
6 print(my_dict)  
  
{'nome': 'Mario', 'cognome': 'Rossi', 'età': 45}  
{'nome': 'Mario', 'cognome': 'Rossi', 'età': 45, 'occupazione': 'Data Analyst'}
```

## Dizionari — metodi

I metodi fondamentali dei dizionari sono:

- ❑ `dict.get(chiave)` : Restituisce il valore associato alla chiave specificata, se presente nel dizionario
- ❑ `dict.items()` : Restituisce una vista di tupla degli elementi del dizionario (coppie chiave-valore)
- ❑ `dict.keys()` : Restituisce una vista degli oggetti chiave nel dizionario
- ❑ `dict.values()` : Restituisce una vista degli oggetti valore nel dizionario
- ❑ `dict.update(iterabile)` : Aggiorna il dizionario con gli elementi forniti nell'iterabile (tupla, lista, o altro dizionario)

## Dizionari — metodi

```
rubrica = {"Marco": "1234", "Anna": "5678", "Andrea": "9012"}

print( rubrica["Anna"] )
>>> 5678

# Un nuovo elemento si aggiunge con l'operatore di assegnazione
rubrica["Serena"] = "3456"

print( rubrica.get("Serena") )
>>> 3456

print( rubrica.keys() )
>>> ["Marco", "Anna", "Andrea", "Serena"]

print( rubrica.values() )
>>> ["1234", "5678", "9012", "3456"]
```

## Dizionari — metodi

```
rubrica = {"Marco": "1234", "Anna": "5678", "Andrea": "9012"}

nuovi_dati = {"Martina": "2002", "Andrea": "1001"}

rubrica.update( nuovi_dati )

print(rubrica)
>>> {"Marco": "1234", "Anna": "5678", "Andrea": "1001", "Martina": "2002"}

coppie = rubrica.values() # Restituisce una LISTA di TUPLE!
print(coppie)
>>> [("Marco", "1234"), ("Anna", "5678"), ("Andrea", "1001"),
      ("Martina", "2002")]

print( coppie[3] )
>>> ("Martina", "2002")
```

## Costrutto logico condizionale

**Un costrutto logico condizionale è un'istruzione che permette di eseguire blocchi di codice in base al verificarsi di una determinata condizione booleana.**

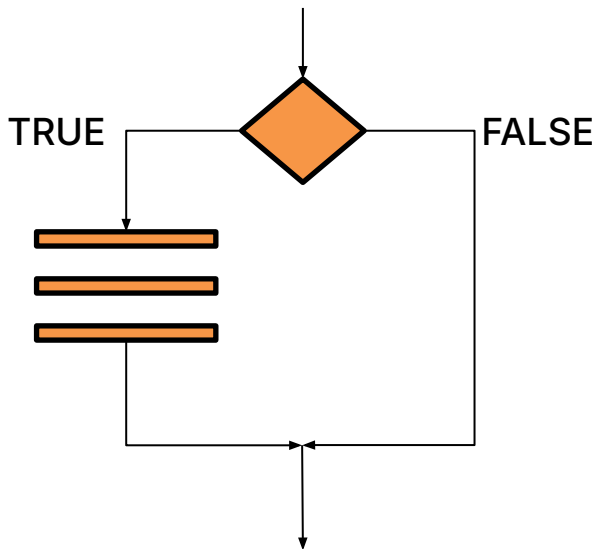




## Costrutto logico condizionale

Capiterà spesso di dover controllare se si verificano determinate condizioni, e di **variare di conseguenza il comportamento del programma**: le istruzioni condizionali servono proprio a questo

La forma più semplice di istruzione condizionale è il **costrutto `if`** ("se" in inglese)



## Costrutto logico condizionale

L'espressione booleana dopo l'istruzione `if` è chiamata **condizione**. Se risulta vera, viene eseguita l'istruzione indentata che segue sulla riga successiva, altrimenti, non succede nulla.

Non c'è limite al numero di istruzioni che possono essere scritte nel corpo, ma deve sempre essercene almeno una.

Sintassi:

`if condizione:`

← **intestazione**

    esegui azione/i

    ← **corpo (con indentazione)**

## Costrutto logico condizionale

Un esempio può essere:

```
1 x = 5
2 y = 7
3 if x < y:
4     print("x è minore di y")
```

x è minore di y

Esempio pratico per un torneo che accetta partecipanti tra i 17 e i 19 anni, per cui vogliamo verificare le età dei partecipanti:

```
1 eta = 17
2 if 17 <= eta <= 19:
3     print("idoneo per il torneo")
```

idoneo per il torneo

## Costrutto logico condizionale

I vari **operatori di confronto** che possiamo usare per definire la condizione sono:

- ❑ **==** (Uguale a): confronta se due valori sono uguali.
- ❑ **!=** (Diverso da): verifica se due valori non sono uguali.
- ❑ **<** (Minore di): controlla se il primo valore è minore del secondo.
- ❑ **>** (Maggiore di): indica se il primo valore è maggiore del secondo.
- ❑ **<=** (Minore o uguale a): verifica se il primo valore è minore o uguale al secondo.
- ❑ **>=** (Maggiore o uguale a): controlla se il primo valore è maggiore o uguale al secondo.
- ❑ **is**: verifica l'identità degli oggetti.
- ❑ **is not**: verifica la non-identità degli oggetti.
- ❑ **in**: verifica se un valore è presente in una sequenza.
- ❑ **not in**: verifica se un valore non è presente in una sequenza.

## Costrutto logico condizionale

Una seconda forma di istruzione `if` è l'**esecuzione alternativa**, dove esistono due possibili azioni, e il valore della condizione determina quale delle due azioni debba essere eseguita e quale no. Dato che la condizione deve essere necessariamente o vera o falsa, sarà sempre eseguita una sola delle due alternative.

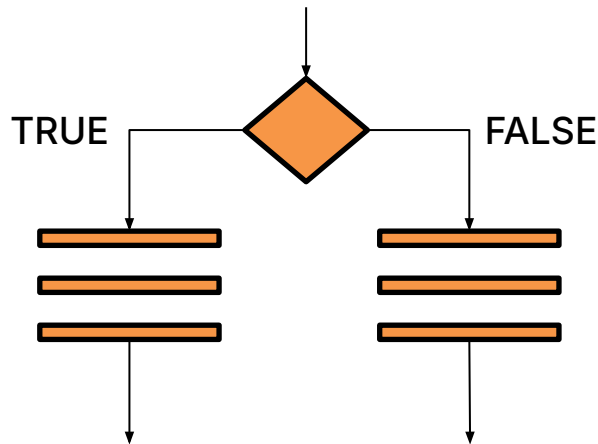
Sintassi:

**if** *condizione*:

    esegui azione/i

**else**:

    esegui altra/e azione/i



## Costrutto logico condizionale

Un esempio può essere:

```
1 x = 5
2 y = 7
3 if x < y:
4     print("x è minore di y")
5 else:
6     print("y è maggiore (o uguale) a x")
```

x è minore di y

O, riutilizzando l'esempio precedente del torneo:

```
1 eta = 20
2 if 17 <= eta <= 19:
3     print("idoneo per il torneo")
4 else:
5     print("non idoneo per il torneo")
```

non idoneo per il torneo

## Costrutto logico condizionale

Verifichiamo che qualcosa si trovi in un insieme, come uno studente in una classe:

```
1 classe = ["Ada", "Ben", "Claude", "Denise", "Emma"]
2 studente = "Claude"
3 if studente in classe:
4     print("Lo studente", studente, "è presente in classe")
5 else:
6     print("Lo studente", studente, "non è presente in classe")
```

Lo studente Claude è presente in classe

Oppure analizziamo i risultati di alcune partite:

```
1 risultati_partite = ["1", "X", "1", "2", "X", "X"]
2 if risultati_partite[-1] == "X":
3     print("L'ultima partita è finita in pareggio")
4 else:
5     print("Nessun pareggio per l'ultima partita")
```

L'ultima partita è finita in pareggio

## Costrutto logico condizionale

### QUIZZONE



Se aggiungessimo l'istruzione:

`risultati_partite.append("2")`

prima del blocco **if**, che cosa accadrebbe?

```
1 risultati_partite = ["1", "X", "1", "2", "X", "X"]
2 risultati_partite.append("2")
3 if risultati_partite[-1] == "X":
4     print("L'ultima partita è finita in pareggio")
5 else:
6     print("Nessun pareggio per l'ultima partita")
```



## Costrutto logico condizionale

A volte è necessario considerare **più di due possibili sviluppi**, e occorre che nel programma ci siano più di due scelte. Un modo per esprimere questo tipo di calcolo sono le condizioni in serie, che seguono la sintassi **if-elif-else**

**if** *condizione*:

    esegui azione/i

**elif** *condizione*:

← anche più di una

    esegui azione/i

**else**:

← opzionale

    esegui azione/i

## Costrutto logico condizionale

- ❑ `elif` è l'abbreviazione di `else if`, che in inglese significa "altrimenti se".
- ❑ Non c'è alcun limite al numero di istruzioni `elif`.
- ❑ Se esiste una clausola `else`, deve essere scritta per ultima, ma non è obbligatoria e viene eseguita solo quando tutte le condizioni precedenti sono false
- ❑ In una "ramificazione" `if/elif/[elif]/[...]/[else]` solo uno dei rami viene eseguito: quindi se una condizione risulta vera, le altre non verranno nemmeno valutate.

## Costrutto logico condizionale

Ad esempio, gestiamo la temperatura di un macchinario industriale:

```
1 temperatura = 27 # °C
2 if temperatura < 20:
3     print("accendere il circuito di riscaldamento")
4 elif temperatura > 30:
5     print("accendere il circuito di raffreddamento")
6 else:
7     print("temperatura operativa ottimale")
temperatura operativa ottimale
```

Oppure, se vogliamo gestire gli studenti iscritti a diversi corsi:

```
1 iscritti_cybersecurity = 10
2 iscritti_data_analyst = 20
3 iscritti_frontend = 5
4 iscritti_backend = 8
5 nuova_iscrizione = "Cybersecurity"
6 if nuova_iscrizione == "Cybersecurity":
7     iscritti_cybersecurity += 1
8 elif nuova_iscrizione == "Data Analyst":
9     iscritti_data_analyst += 1
10 elif nuova_iscrizione == "Frontend":
11     iscritti_frontend += 1
12 elif nuova_iscrizione == "Backend":
13     iscritti_backend += 1
```

## Loops

**Un loop è una struttura di controllo che permette di eseguire ripetutamente un blocco di istruzioni fino a quando una determinata condizione risulta vera.**



## Loops

Spesso i computer sono usati per **automatizzare dei compiti ripetitivi**: ripetere operazioni identiche o simili senza fare errori, è qualcosa che i computer fanno molto bene e le persone piuttosto male.

In programmazione, la ripetizione è chiamata anche **iterazione**.

Python fornisce due modalità di iterazione:

costrutto **while**

costrutto **for**

## While loop

`while` in inglese significa "finché". In particolare, **finché è vera una determinata condizione, il costrutto esegue sempre le stesse istruzioni**, eseguendo quello che viene chiamato ciclo (loop)

Sintassi:

**while** *condizione*:

← **intestazione**

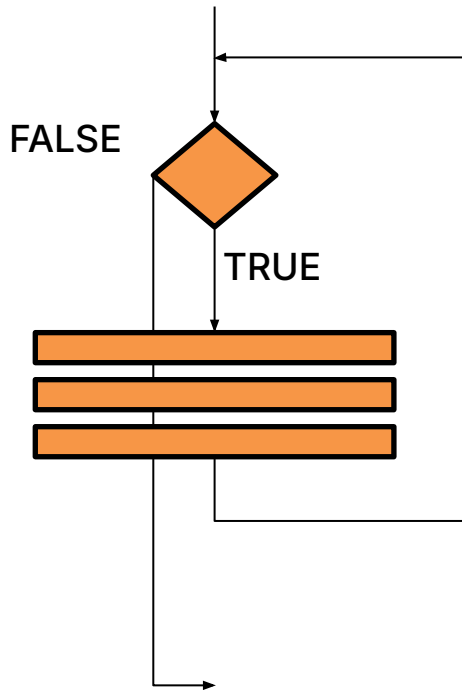
    esegui azione/i

← **corpo (con indentazione)**

## While loop

Il corpo del ciclo `while` deve cambiare il valore di una o più variabili in modo che la condizione prima o poi diventi falsa e il ciclo abbia termine, altrimenti il ciclo verrebbe ripetuto continuamente, dando luogo ad un ciclo infinito.

Potrebbe anche accadere che la condizione sia falsa fin dall'inizio: in tal caso il codice contenuto nel corpo non verrà eseguito neanche una volta.



## Loops

Ad esempio, sommiamo i primi cinque numeri:

```
1 num = 1    # inizializzazione
2 somma = 0  # inizializzazione
3 while num <= 5:
4     somma += num # esecuzione del calcolo
5     print("Num ora è: ", num, "e la somma ora è: ", somma)
6     num += 1 # ci assicura che il ciclo finisca
7 print("La somma finale dei primi 5 numeri è:", somma)
```

```
Num ora è:  1 e la somma ora è:  1
Num ora è:  2 e la somma ora è:  3
Num ora è:  3 e la somma ora è:  6
Num ora è:  4 e la somma ora è: 10
Num ora è:  5 e la somma ora è: 15
La somma finale dei primi 5 numeri è: 15
```



## Loops

Nell'esempio precedente abbiamo:

- ❑ **Inizializzato delle variabili** (*num* e *somma*): sono quelle che serviranno per le operazioni da svolgere e/o per verificare la condizione
- ❑ **Definito l'intestazione del costrutto `while` con la condizione che verrà valutata ad ogni iterazione** e sulla base della quale verrà oppure non verrà eseguito il codice indentato sottostante (*num* ≤ 5)
- ❑ **Ricalcolato la somma** (*somma*) aggiungendo il numero (*num*)
- ❑ **Stampato** il valore del numero (*num*) **per come è in quel momento del processo** e il valore della somma (*somma*)
- ❑ **Incrementato il valore del numero** (*num*)
- ❑ **Ripetuto i precedenti step fino a quando la condizione si rivela falsa**, ovvero quando il valore della variabile *num* non è più minore o uguale di 5 (quindi quando è maggiore di 5) usciamo dal loop ed eseguiamo l'ultimo `print()`

## Loops

Esempio, chiediamo all'utente di inserire dei numeri, e sommiamoli finché il valore della somma non esce dall'intervallo tra 0 e 50:

```
1 somma = 0 # inizializzazione
2 num = input("Inserisci un numero ") # prendiamo un numero in input dall'utente
3 while 0 <= somma <= 50:
4     num = int(num) # casting
5     somma += num
6 print("La somma è:", somma)
```

```
Inserisci un numero20
La somma è: 60
```

Qui stiamo usando anche la funzione `input()` per interagire con l'utente, e stiamo cambiando il tipo di dato di una variabile (riga 4) tramite un'operazione che si chiama **type casting**.

## Loops

Possiamo usare un costrutto while per calcolare i primi 10 numeri della sequenza di Fibonacci:

```
1 numero1 = 1    # inizializzazione
2 numero2 = 1    # inizializzazione
3 fibo = [1, 1]  # inizializzazione - conterrà i risultati
4 conteggio = 3  # inizializzazione - già abbiamo i primi due!
5 while conteggio <= 10:
6     prossimo_numero = numero1 + numero2 # calcolo
7     fibo.append(prossimo_numero) # raccolta dei risultati
8     numero1 = numero2             # aggiornamento
9     numero2 = prossimo_numero      # aggiornamento
10    conteggio += 1 # ci assicura che il ciclo finisca
11 print("I primi 10 numeri di Fibonacci sono:", fibo)
```

I primi 10 numeri di Fibonacci sono: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

## Costrutto logico condizionale

### QUIZZONE



Cosa succede se invertiamo le istruzioni di aggiornamento da:

numero1 = numero2

numero2 = prossimo\_numero

a:

numero2 = prossimo\_numero

numero1 = numero2

```
1 numero1 = 1
2 numero2 = 1
3 fibo = [1, 1]
4 conteggio = 3
5 while conteggio <= 10:
6     prossimo_numero = numero1 + numero2
7     fibo.append(prossimo_numero)
8     numero2 = prossimo_numero
9     numero1 = numero2
10    conteggio += 1
11 print("I primi 10 numeri di Fibonacci sono:", fibo)
```

## Costrutto logico condizionale

### QUIZZONE



Cosa succede se eliminiamo l'istruzione:

`conteggio += 1`

```
1 numero1 = 1
2 numero2 = 1
3 fibo = [1, 1]
4 conteggio = 3
5 while conteggio <= 10:
6     prossimo_numero = numero1 + numero2
7     fibo.append(prossimo_numero)
8     numero1 = numero2
9 print("I primi 10 numeri di Fibonacci sono:", fibo)
```

## For loop

Il **costrutto for** si basa sul fatto di andare a considerare, uno per uno, tutti gli elementi di un oggetto che contiene per l'appunto più elementi, quali liste, tuple, set, stringhe o dizionari.

A differenza del costrutto `while`, qui sappiamo già in principio quante volte dovremo effettuare il ciclo.

Si adatta perfettamente a quei tipi di calcolo che comportano l'elaborazione di una collection, un elemento per volta, iniziando dal primo elemento, eseguendo una certa operazione e continuando fino alla fine della collezione. Questo tipo di elaborazione è detta **attraversamento**.

## For loop

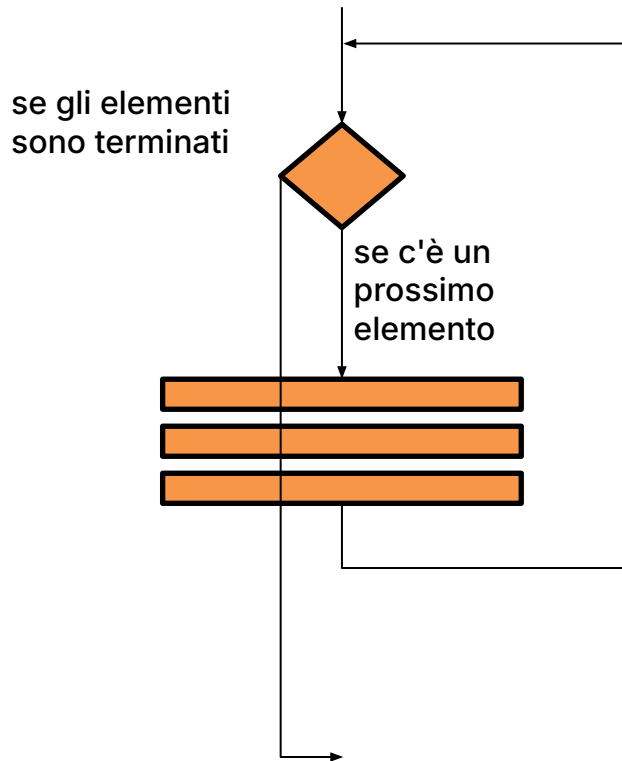
`for` in inglese significa "per", nel senso di "per ognuno [degli elementi]".

Esegue lo stesso pezzo di codice per ognuno degli elementi in una collezione, dunque iterando su ognuno di essi.

Sintassi:

**for** *elemento in iterabile*:

    esegui azione/i sull'elemento



## For loop

Possiamo gestire un attraversamento tramite un ciclo `while` e l'indicizzazione:

```
1 lista_prezzi = [10, 20, 30, 40, 50]
2 indice = 0 # inizializzazione
3 while indice < len(lista_prezzi):
4     prezzo = lista_prezzi[indice]
5     print("-", prezzo)
6     indice = indice + 1
```

```
- 10
- 20
- 30
- 40
- 50
```

Ma è molto più efficace, in questi casi, usare un **costrutto `for`**.

```
1 lista_prezzi = [10, 20, 30, 40, 50]
2 for prezzo in lista_prezzi:
3     print("-", prezzo)
4
```

```
- 10
- 20
- 30
- 40
- 50
```



## For loop

Vediamo alcuni esempi:

```
1 insieme = 1, 4, 6, 7, 2, 8
2 for numero in insieme:
3     print(numero * 2)
```

```
2
8
12
14
4
16
```

```
1 stipendi = [100, 200, 300, 400]
2 for stipendio in stipendi:
3     print(stipendio + 10)
```

```
110
210
310
410
```

```
1 parola = "Python"
2 for lettera in parola:
3     print(lettera + "!")
```

```
P!
y!
t!
h!
o!
n!
```

## For loop

Esempio pratico: applichiamo uno sconto ai prezzi dei prodotti di una catena di negozi:

```
1 prezzi = (20, 40, 50, 60, 190, 30)
2 prezzi_scontati = [] # inizializzazione
3 for prezzo in prezzi:
4     print("Prezzo originale:", prezzo)
5     prezzi_scontati.append(prezzo * 0.8)
6     print("Prezzo scontato:", prezzo * 0.8)
7     print("")
8 print("Finito.")
9
```

```
Prezzo originale: 20
Prezzo scontato: 16.0

Prezzo originale: 40
Prezzo scontato: 32.0

Prezzo originale: 50
Prezzo scontato: 40.0

Prezzo originale: 60
Prezzo scontato: 48.0

Prezzo originale: 190
Prezzo scontato: 152.0

Prezzo originale: 30
Prezzo scontato: 24.0

Finito.
```

## For loop

Aggiungiamo una variabile per cambiare sconto a seconda della necessità:

```
1 prezzi = (20, 40, 50, 60, 190, 30)
2 sconto = 0.2
3 prezzi_scontati = [] # inizializzazione
4 for prezzo in prezzi:
5     print("Prezzo originale:", prezzo)
6     prezzo_scontato = prezzo * (1-sconto)
7     prezzi_scontati.append(prezzo_scontato)
8     print("Prezzo scontato:", prezzo_scontato)
9     print("")
10 print("Finito.")
```

```
Prezzo originale: 20
Prezzo scontato: 16.0
```

```
Prezzo originale: 40
Prezzo scontato: 32.0
```

```
Prezzo originale: 50
Prezzo scontato: 40.0
```

```
Prezzo originale: 60
Prezzo scontato: 48.0
```

```
Prezzo originale: 190
Prezzo scontato: 152.0
```

```
Prezzo originale: 30
Prezzo scontato: 24.0
```

```
Finito.
```

## For loop

Usiamo due percentuali di sconto diverse a seconda dell'entità del prezzo:

```
1 prezzi = (20, 40, 50, 60, 190, 30)
2 sconto1, sconto2 = 30, 10 # %
3 prezzi_scontati = [] # inizializzazione
4 for prezzo in prezzi:
5     print("Prezzo originale:", prezzo)
6     if prezzo > 55:
7         prezzo_scontato = prezzo * (1 - sconto1/100)
8         print("Prezzo scontato del", sconto1, "%:", prezzo_scontato)
9     else:
10        prezzo_scontato = prezzo * (1 - sconto2/100)
11        print("Prezzo scontato del", sconto2, "%:", prezzo_scontato)
12    print("")
13    prezzi_scontati.append(prezzo_scontato)
14 print("Finito, ecco tutti i prezzi scontati:", prezzi_scontati)
15
```

Prezzo originale: 20  
Prezzo scontato del 10 %: 18.0

Prezzo originale: 40  
Prezzo scontato del 10 %: 36.0

Prezzo originale: 50  
Prezzo scontato del 10 %: 45.0

Prezzo originale: 60  
Prezzo scontato del 30 %: 42.0

Prezzo originale: 190  
Prezzo scontato del 30 %: 133.0

Prezzo originale: 30  
Prezzo scontato del 10 %: 27.0

Finito, ecco tutti i prezzi scontati: [18.0, 36.0, 45.0, 42.0, 133.0, 27.0]



**GRAZIE**  
Epicode