PARTE 3

Strutture di controllo

Nicolò Bartolini

Argomenti

Bonus: funzione input() e type casting

Le strutture di controllo

Condizionali (if, else, elif)

Ciclo while

→ **Bonus: funzione input() e type casting**

Le strutture di controllo

Condizionali (if, else, elif)

Ciclo while

Le funzioni ritornano sempre un valore

Ritornare significa che la funzione restituisce un valore, quindi è come se al posto del punto in cui la funzione viene chiamata ci fosse il risultato della sua esecuzione. L'interprete esegue il codice della funzione e utilizza il risultato "per sostituire" il punto in cui viene chiamata.

Ogni funzione in Python **ritorna sempre un valore**. Questo concetto è fondamentale per comprendere appieno il modo in cui le funzioni devono essere utilizzate.

Abbiamo visto che quando scriviamo noi una funzione, possiamo specificare cosa ritornerà questa funzione attraverso la parola chiave `return`.

```
1   def somma(a, b):
2       return a + b
```

La funzione `somma` prende due argomenti `a` e `b` e ritorna la loro somma

Nel codice qui sotto, la variabile x conterrà il valore 3, che è il *valore di ritorno* della funzione `somma` con i parametri a e b impostati rispettivamente a 1 e 2. La cosa da comprendere è che la variabile x contiene quel risultato, non la funzione stessa.

```
x = somma(1, 2)
```

```
7   somma(1, 2)
```

Se non si salva il risultato di una funzione in una variabile, invece, il suo valore di ritorno viene **perso per sempre** e diventa **inutilizzabile**, come nel secondo esempio qui a sinistra.

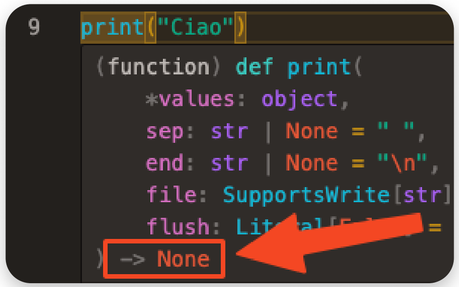
N.B.: l'istruzione return viene utilizzata soltanto dentro le funzioni create da noi, per specificare cosa ritorna la funzione. Non deve essere usata fuori.

Tipo di ritorno

E se volessimo creare una funzione che fa solo delle cose senza restituire niente?

La risposta è che si può fare. Altri linguaggi obbligano l'inserimento della parola chiave `return`, ma Python no.

Quando una funzione, in Python, non possiede al suo interno un'istruzione `return`, allora questa funzione, idealmente, **non restituisce nulla**. Il nulla in Python, però, è il valore None, di tipo `NoneType`. Quindi anche se la funzione non restituisce nulla, **in realtà restituisce None**. *In questi casi **ha senso** non assegnare il risultato della funziona a una variabile.*



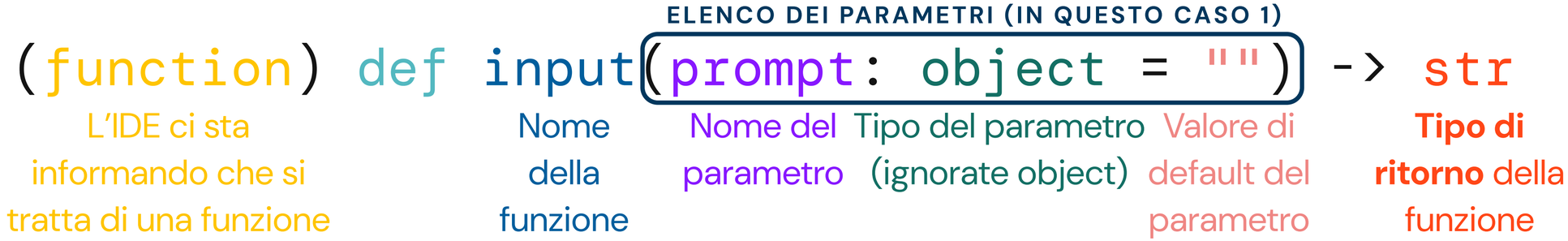
Un esempio di funzione che non restituisce nulla è la funzione built-in `print()`. Questo tipo di funzioni eseguono delle operazioni (`print()` stampa qualcosa nello schermo) ma poi non restituiscono alcun valore. Come possiamo vedere nella figura, l'IDE ci aiuta a capire qual è il **tipo di ritorno** di una funzione portandoci il mouse sopra.

Il concetto di **tipo di ritorno** è fondamentale e fornisce un'ulteriore utilità ai già importantissimi **tipi di dati** introdotti nella parte precedente. Infatti, le funzioni, generalmente, possiedono un tipo di ritorno che indica **di che tipo sarà il valore che viene restituito dalla funzione stessa**. Questa cosa è fondamentale per capire **come si deve lavorare** con il dato che restituisce la funzione. Ad esempio, se una funzione restituisce un booleano, non si potrà usare per fare un'addizione.

La funzione built-in `input()`

La funzione built-in `input()` è una funzione che permette di introdurre dell'**interattività** all'interno dei propri programmi in Python. Il suo scopo è quello di **raccogliere dati dall'utente**.

Introduciamo la **firma** delle funzioni, attraverso la quale possiamo comprendere come utilizzarle. Portando il mouse su una funzione built-in, l'IDE ci fornisce delle informazioni nella seguente forma:



Quindi, la funzione `input()`, è una funzione che accetta un unico parametro chiamato `prompt`, di tipo `object` (per ora consideriamo che sia di tipo stringa), il quale possiede un **valore di default** pari a `""`, ovvero una *stringa vuota* e che come **tipo di ritorno** ha `str`, quindi **restituisce sempre un valore di tipo stringa**.

Quando un parametro di una funzione possiede un valore di default, significa che quel parametro **può essere omesso** quando la funzione viene chiamata e, se viene omesso, assume automaticamente quel valore.

Utilizzo della funzione `input()`

Come abbiamo detto, la funzione built-in `input()` viene utilizzata per raccogliere dei dati dall'utente. Abbiamo anche visto che possiede un parametro opzionale chiamato `prompt`. Questo prompt serve per stampare sul terminale una stringa a propria scelta. Dunque, quello che fa la funzione `input()` è stampare l'eventuale `prompt` (o non stampare nulla se prompt non viene specificato) e poi mettersi in ascolto di qualcosa scritto dall'utente. Quando l'utente scrive qualsiasi cosa e poi clicca invio, allora la funzione input() restituisce una stringa contenente ciò che ha scritto l'utente.

Operatore di assegnamento

```
1   qualcosa_ottenuto_da_utente = input("Ciao utente, inserisci qualcosa: ")
```

Creiamo una variabile chiamata qualcosa_ottenuto_da_utente

Inseriamo dentro la variabile **il valore che viene restituito** dalla funzione `input()` chiamata con il parametro `prompt` pari a `"Ciao utente, inserisci qualcosa: "`

Eseguendo questo codice, nel terminale comparirà:

```
Ciao utente, inserisci qualcosa:
```

Lo spazio vuoto alla fine della stringa viene utilizzato per motivi estetici. Omettendolo, infatti, il testo inserito dall'utente sarebbe appiccicato al prompt.

Il rettangolo finale (nel mio caso giallo, ma nel vostro potrebbe essere diverso) indica che **il programma è in attesa di qualcosa inserito dall'utente**. L'inserimento di quel qualcosa e il clicco su "Invio" farà proseguire il codice e inserirà la **stringa** contenente ciò che ha inserito l'utente all'interno della **variabile** qualcosa_ottenuto_da_utente.

Type casting

Nella slide 6, abbiamo analizzato la **firma** della funzione `input()`, osservando che ha come **tipo di ritorno** `str`. Ciò significa che tutto ciò che restituisce la funzione `input()` è sempre di tipo `str`. Utilizziamo questa caratteristica per introdurre il concetto di **type casting** (o **casting di tipo**).

Supponiamo di voler far inserire dall'utente due numeri e di volerli sottrarre. Per fare ciò utilizziamo due volte la funzione `input()` chiedendo due numeri all'utente e poi printiamo la loro differenza, come nel codice seguente.

```
1   numero1 = input("Inserisci il primo numero: ")
2   numero2 = input("Inserisci il secondo numero: ")
3   print("La sottrazione di ", numero1, " e ", numero2, " è ", numero1 - numero2)
```

Osserviamo, però, che l'esecuzione del codice porta a un **errore di tipo**, indicando che l'interprete non riesce ad eseguire una *sottrazione tra due valori di tipo* `str`.

```
Inserisci il primo numero: 3
Inserisci il secondo numero: 1
Traceback (most recent call last):
  File "/Users/nickb/Dev/prova.py", line 3, in <module>
    print("La sottrazione di ", numero1, " e ", numero2, " è ", numero1 - numero2)
                                                               ~~~~~~~~^~~~~~~~~~
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Per risolvere queste problematiche si effettua il cosiddetto **type casting**, ovvero una **conversione forzata** di un valore in uno specifico tipo.
Ovviamente, il valore che si vuole convertire deve essere convertibile. Ad esempio, **non** si può convertire la stringa *"quaranta"* in un `int`, così come non si può convertire la stringa *"40 1"*. Invece, si può convertire la stringa *"12"*.

C'è uno spazio in mezzo.

Come si effettua il type casting in Python?

Per effettuare operazioni di **type casting** (d'ora in avanti lo chiameremo solo "**casting**"), Python fornisce delle **funzioni built-in** che si chiamano come il tipo verso cui effettuare il casting.

- La funzione `int(da_convertire)` restituisce il valore `da_convertire` convertito in tipo `int`. Il valore da convertire può essere un `int` (restituisce lo stesso numero), un `float` (restituisce lo stesso numero senza parte decimale) o una `str` (cerca di verificare la presenza di numeri all'interno della stringa e se li trova li restituisce in tipo `int`).
- La funzione `float(da_convertire)` restituisce il valore `da_convertire` convertito in tipo `float`. Il comportamento è analogo a quello della funzione `int()`.
- La funzione `str(da_convertire)` restituisce il valore `da_convertire` convertito in tipo `str`. Il valore da convertire può essere qualsiasi tipo di valore, quindi un `int`, un `float`, una `str`, un `bool`, un `None`, ma anche altri tipi che scopriremo più avanti.
- La funzione `bool(da_convertire)` restituisce il valore `da_convertire` convertito in tipo `bool`. Questa funzione restituisce sempre `True`, tranne quando `da_convertire` è un int pari a `0`, un float pari a `0.0`, una stringa vuota (`""`) oppure un `None`, casistiche in cui restituisce `False`.

```
10  int(1)  # Output: 1
11  int(1.2)  # Output: 1
12  int("1")  # Output: 1
13  int("   1\n")  # Output: 1
14  int("1.2")  # Errore
15  int("12cm")  # Errore
```

```
17  float(1)  # Output: 1.0
18  float("1.2")  # Output: 1.2
19  float("12.7cm")  # Errore
```

```
21  str(1)  # Output: "1"
22  str(1.2)  # Output: "1.2"
23  str("ciao")  # Output: "ciao"
24  str(True)  # Output: "True"
25  str(None)  # Output: "None"
```

```
27  bool(259)  # Output: True
28  bool("Ao!")  # Output: True
29  bool(0)  # Output: False
30  bool(None)  # Output: False
31  bool("")  # Output: False
```

Come risolvere il problema di `input()`?

Ora che abbiamo introdotto il casting, come possiamo risolvere il **problema** che abbiamo incontrato nella slide 8 con la funzione `input()`?

La risposta risiede, ovviamente, nel **castare** le variabili che contengono il risultato della funzione `input()` a un **tipo numerico**. Scegliamo, in modo casuale, di voler lavorare con i `float`. Dunque, possiamo utilizzare la funzione `float()` per castare in `float` i valori inseriti dall'utente, che sono restituiti da `input()` sotto forma di `str`. Il codice, diventa:

```python
numero1 = input("Inserisci il primo numero: ")
numero2 = input("Inserisci il secondo numero: ")

numero1 = float(numero1)  # Oppure numero1 = int(numero1)
numero2 = float(numero2)  # Oppure numero2 = int(numero2)

print("La sottrazione di ", numero1, " e ", numero2, " è ", numero1 - numero2)
```

Per rendere il tutto più compatto, si può anche passare il risultato di **`input()`** direttamente alla funzione **`float()`** e inserire il risultato della funzione **`float()`** direttamente durante la creazione delle variabili, nel seguente modo:

```python
numero1 = float(input("Inserisci il primo numero: "))
numero2 = float(input("Inserisci il secondo numero: "))
print("La sottrazione di ", numero1, " e ", numero2, " è ", numero1 - numero2)
```

→

```
Inserisci il primo numero: 2
Inserisci il secondo numero: 3.6
La sottrazione di  2.0  e  3.6  è  -1.6
```

Esempio dell'utilizzo di `input()`

PROBLEMA: Scrivere un programma che effettua semplici calcoli. Il programma deve:

1. Scrivere un messaggio di benvenuto all'esecuzione, chiedendo il nome dell'utente e poi salutarlo.
2. Chiedere all'utente tre numeri qualsiasi.
3. Sommare i tre numeri e mostrare il risultato all'utente.
4. Chiedere un quarto numero all'utente.
5. Sottrarlo alla somma fatta precedentemente e stampare il risultato di tale sottrazione.
6. Salutare l'utente.

Note e consigli aggiuntivi:

- Commentare adeguatamente il codice per esplicare chiaramente tutti i passaggi svolti. Questa richiesta è utile per abituarsi a scrivere commenti nel codice, molto importanti.
- Utilizzare dei print() intermedi che non sono richiesti nella consegna principale, se necessario. Questi print() possono tornare molto utili quando non si capisce perché il programma si comporta in modo anomalo.
- Rendere il programma "accessibile", spiegando chiaramente, attraverso le stampe, quello che si richiede all'utente e quello che si mostra all'utente.
- Si suppone che l'utente non si comporti in modo anomalo e che quindi, se gli viene richiesto di inserire un numero intero, egli inserisca davvero un numero intero.

Soluzione dell'esempio

N.B.: Vi ricordo che se un problema è risolvibile, lo è in infiniti modi. Quindi ci sono infinite soluzioni corrette. Questa è solo la mia soluzione, una tra le infinite possibili.

Ricordiamoci che un programma è una **sequenza di istruzioni**. Quindi, svolgiamo l'esercizio ragionando istruzione per istruzione leggendo, **step dopo step**, la consegna.

"Step 1: Il programma deve (1.1) scrivere un messaggio di benvenuto all'esecuzione, chiedendo il nome dell'utente e poi (1.2) salutarlo."

Abbiamo suddiviso lo step 1 della consegna in due **sotto-step**, che sono le possibili istruzioni per Python. Per implementare lo step 1.1 possiamo utilizzare la funzione `input()` sia per dare il benvenuto all'utente che per memorizzare il suo nome. Il codice è, quindi, il seguente:

```
# Diamo il benvenuto all'utente e chiediamo il suo nome
nome_utente = input("Benvenuto! Qual è il tuo nome? ")
```

Per quanto riguarda lo step 1.2, è sufficiente utilizzare la funzione `print()` per salutare l'utente utilizzando il suo nome, che abbiamo memorizzato nella variabile `nome_utente`.

```
print("Ciao", nome_utente, "!")  # Oppure: print("Ciao " + nome_utente + "!")
```

Notiamo l'attenzione posta nell'"**accessibilità**", stampando delle stringhe che siano ben formate e ben scritte (es.: maiuscole corrette e spazi vuoti alla fine degli input), caratteristiche che sono inutili ai fini del nostro esercizio, ma che è bene apprendere per scrivere programmi che non facciano sentire l'utente come un pesce fuor d'acqua.

Soluzione dell'esempio

N.B.: Vi ricordo che se un problema è risolvibile, lo è in infiniti modi. Quindi ci sono infinite soluzioni corrette. Questa è solo la mia soluzione, una tra le infinite possibili.

"Step 2: Il programma deve chiedere all'utente tre numeri qualsiasi."

In questo caso non abbiamo sotto-step, quindi possiamo direttamente tradurre la consegna in Python.

```python
# Ora chiediamo all'utente di inserire tre numeri
print("Bene, ", nome_utente, ", ora inserisci tre numeri qualsiasi.")
# Otteniamo i tre numeri attraverso la funzione input e castiamo il
# risultato della funzione a float per evitare errori di tipo
num1 = float(input())  # Notiamo che abbiamo già chiesto all'utente
num2 = float(input())  # di inserire tre numeri, quindi non inseriamo
num3 = float(input())  # un prompt negli input. Ma è una scelta stilistica.
```

Come è specificato anche nei commenti, io ho scelto di printare una stringa che chiede all'utente **direttamente** di inserire tre numeri qualsiasi. Per questo motivo mi è sembrato **superfluo** chiedere nuovamente il numero da inserire nel prompt di ogni `input`. Tuttavia, questa è una scelta totalmente **stilistica** che non intacca minimamente il funzionamento del programma e il raggiungimento dell'obiettivo.

Un'altra cosa che può essere fatta in modo diverso è il **casting**. Infatti, io ho effettuato il casting della stringa ottenuta dagli `input()` a `float` passando direttamente la funzione `input()` alla funzione `float()`. Come abbiamo visto nella slide 10, si può fare anche **separatamente** senza intaccare il funzionamento del programma.

Soluzione dell'esempio

N.B.: Vi ricordo che se un problema è risolvibile, lo è in infiniti modi. Quindi ci sono infinite soluzioni corrette. Questa è solo la mia soluzione, una tra le infinite possibili.

"Step 3: Il programma deve (3.1) sommare i tre numeri e (3.2) mostrare il risultato all'utente."

Possiamo risolvere lo step 3.1 in una riga e lo step 3.2 in un'altra. Il codice è il seguente:

```
# Sommiamo i tre numeri e stampiamo il risultato di questa somma
somma = num1 + num2 + num3
print("La somma dei tre numeri è", somma)
```

"Step 4: Il programma deve chiedere un quarto numero all'utente."

```
# Ora chiediamo all'utente di inserire un ulteriore numero
num4 = float(input("Bene " + nome_utente + ", ora inserisci un ulteriore numero: "))
```

Notiamo, nell'`input()` qui sopra, che abbiamo effettuato la **concatenazione** della stringa `prompt`, non attraverso le virgole (come nei `print`), ma attraverso la **somma di stringhe**. Questo perché la concatenazione di stringhe tramite virgole è disponibile solo nella funzione `print()` e non nella funzione `input()`.
Osservazione importante: se avessimo dovuto inserire tra le stringhe una variabile contenente un **numero**, avremmo dovuto **castarlo a stringa** con la funzione `str()`. Ad esempio:

```
num4 = float(input("La somma dei tre numeri è " + str(somma) + ", ora inserisci un ulteriore numero: "))
```

Soluzione dell'esempio

N.B.: Vi ricordo che se un problema è risolvibile, lo è in infiniti modi. Quindi ci sono infinite soluzioni corrette. Questa è solo la mia soluzione, una tra le infinite possibili.

"Step 5: Il programma deve (5.1) sottrarlo alla somma fatta precedentemente e (5.2) stampare il risultato di tale sottrazione."

Anche in questo caso, la soluzione è rapida:

```
# Sottraiamo il nuovo numero dalla somma e stampiamo il risultato
risultato = somma - num4
print("Il risultato della sottrazione è", risultato)
```

Tuttavia, visto che non dobbiamo più utilizzare questo risultato in parti successive del codice, possiamo evitare la creazione di una nuova variabile e risolvere questo step in un'unica riga:

```
print("Il risultato della sottrazione è", somma - num4)
```

"Step 6: Il programma deve salutare l'utente."

Dopo tutti gli step precedenti, questo è un gioco da ragazzi:

```
# Salutiamo l'utente come istruzione finale
print("Ciao", nome_utente, ", è stato un piacere fare questi calcoli.")
```

Altri esercizi

Propongo altri **esercizi** da svolgere sull'argomento `input()` e casting. Sono sempre la "stessa solfa" di quello appena descritto, però svolgerli aiuta ad abituarsi a scrivere codice e a capire meglio come funziona il tutto.

① *Scrivere un programma che permetta di convertire un valore in una specifica unità di misura in un'altra unità di misura. Le unità di misura di partenza e di arrivo devono essere predeterminate, ovvero si chiede all'utente di inserire un valore, ad esempio, in miglia e il programma lo converte in kilometri. Dunque, il programma non deve funzionare in base all'unità di misura inserita dall'utente, ma è il programma stesso a specificare l'unità di misura richiesta.*

② *Scrivere un programma che permetta di calcolare l'area di un triangolo (o di un'altra forma geometrica). Il programma deve reperire dall'utente i dati di cui ha bisogno, effettuare il calcolo e mostrare il risultato.*

③ *Scrivere un programma che permetta di calcolare il valore futuro di un investimento. Il programma deve accettare, da parte dell'utente, il quantitativo di denaro da investire, il tasso di interesse e il numero di anni di durata dell'investimento. Il programma deve calcolare il quantitativo di denaro che si avrà dopo il numero di anni inserito con il tasso di interesse inserito.*

N.B.: Ogni stampa nel terminale deve essere **esplicativa** e **chiara**. Ad esempio, è meglio non eseguire `print(area_triangolo)`, ma eseguire `print("L'area del triangolo è: ", area_triangolo)`.

Bonus: funzione input() e type casting

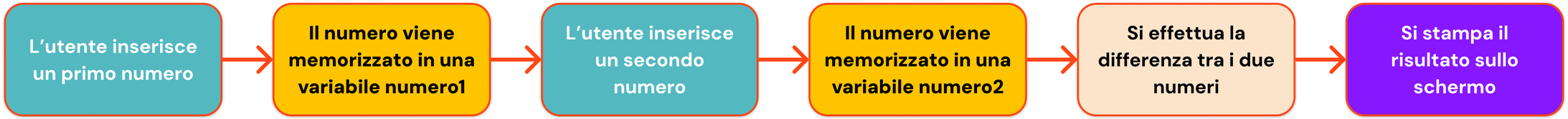
→ **Le strutture di controllo**

Condizionali (if, else, elif)

Ciclo while

Flusso di esecuzione di un programma

Finora abbiamo visto programmi che eseguono un elenco di istruzioni **sequenzialmente**, dalla prima all'ultima. Il modo in cui un programma esegue le istruzione è chiamato **flusso di esecuzione**. Il flusso che abbiamo incontrato finora è quello schematizzato nell'immagine seguente:



Tuttavia, anche considerando gli esercizi svolti e proposti nella sezione precedente, è evidente che con questo **approccio limitato**, le opzioni per la creazione di programmi sono estremamente limitate. In pratica, poiché il **flusso** del programma è **unidirezionale**, tutti i programmi scritti in questo modo risultano essere sostanzialmente **simili**, seppur contestualizzati in modi differenti.

Naturalmente, il processo di programmazione non si limita a questo punto. Al contrario, questo rappresenta appena l'**inizio** di un lungo percorso. A questo punto entrano in gioco le **strutture di controllo**, che consentono di **modificare radicalmente** il funzionamento finora esaminato e di aprire le porte a innumerevoli nuove opportunità di sviluppo.

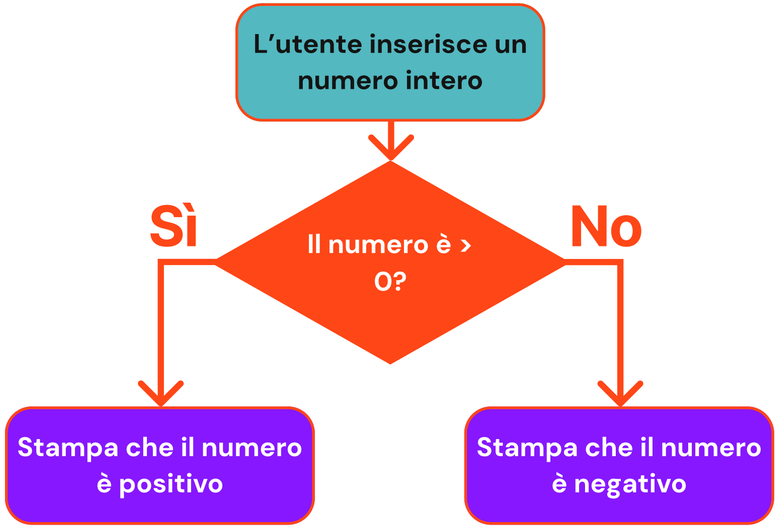
Le strutture di controllo

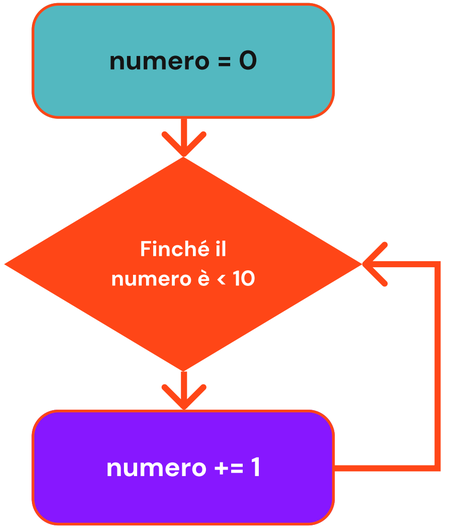
Le **strutture di controllo** sono elementi fondamentali della programmazione. Esse consentono di **modificare** il normale **flusso di esecuzione** di un programma, indirizzandolo verso **percorsi diversi** da quello tradizionale finora esaminato. Questo significa che possono influenzare significativamente il comportamento del software.

Le strutture di controllo si suddividono principalmente in due categorie: **condizionali** e **cicli**. Le strutture **condizionali** modificano il flusso del programma in base al **valore di verità** di condizioni specificate (booleani). I **cicli**, invece, consentono al programma di **iterare** (ripetere più volte) su un insieme di istruzioni, ripetendo l'esecuzione fino al soddisfacimento di determinate **condizioni di uscita**.

In Python, la struttura condizionale è l'`if`, arricchito con le istruzioni `else` ed `elif`.

I cicli di Python, invece, sono il ciclo `while` e il ciclo `for`.





Bonus: funzione input() e type casting

Le strutture di controllo

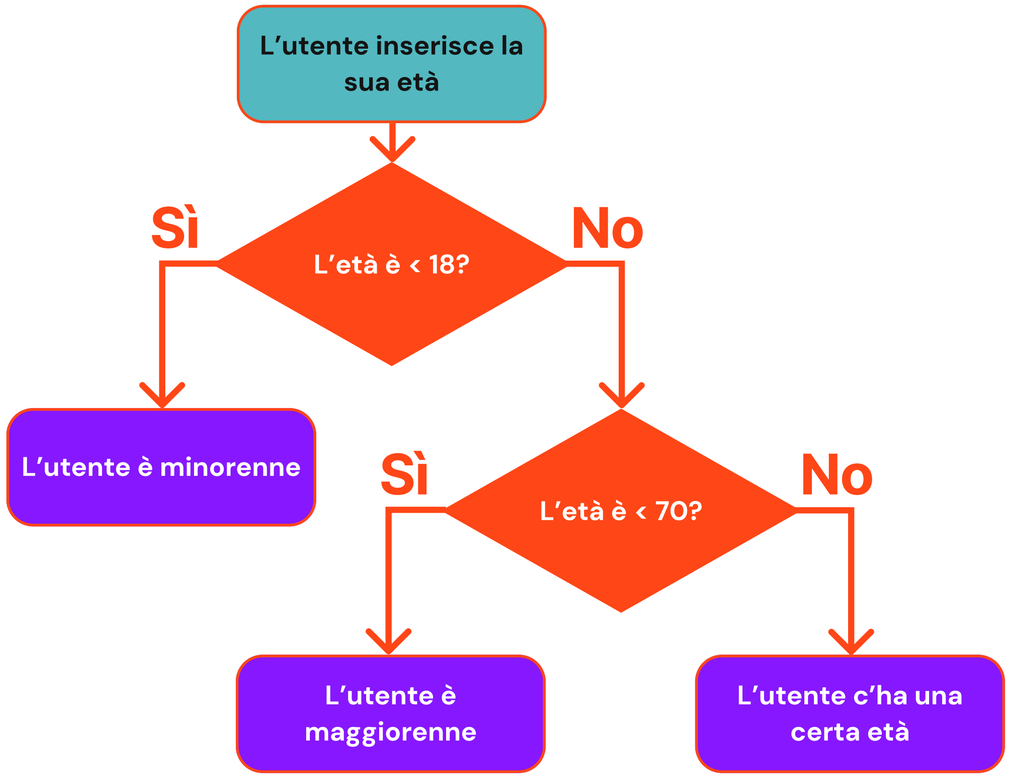
→ **Condizionali (if, else, elif)**

Ciclo while

Le strutture condizionali

Le **strutture condizionali** sono gli strumenti essenziali che permettono al programma di *"prendere decisioni"*. A livello tecnico, le strutture condizionali eseguono **diversi blocchi di codice** in base al **valore di verità** di una specifica **condizione**. In altre parole, in base al valore **booleano** della condizione (`True` o `False`), il blocco di codice corrispondente viene o non viene eseguito.

Il **vantaggio** dell'utilizzo delle strutture condizionali è che permette di **rispondere dinamicamente** agli input che, per definizione, sono **dinamici**. Un ulteriore vantaggio è che il codice, con l'utilizzo delle strutture condizionali, è ovviamente più **intelligente** e **flessibile** e si adatta a molteplici situazioni, piuttosto che seguire un ordine preciso, prestabilito e **immutabile**.

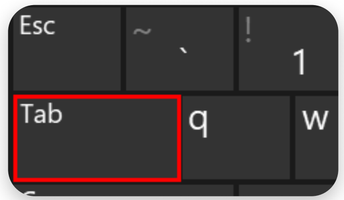


L'istruzione `if`

L'istruzione `if` verifica il **valore di verità** di una specifica **condizione** e, se tale valore risulta vero (`True`), allora esegue il **blocco di codice** corrispondente.
La sintassi è la seguente:

```
1 if condizione:
2     <blocco di codice da eseguire se la condizione è True>
```

Notiamo che sono presenti 4 spazi (una tabulazione) che, come ricordiamo dal modo in cui si creano le funzioni, definiscono, in Python, un blocco di codice indentato.



Una **condizione** è un'**espressione** che verifica la **verità** di qualcosa, generalmente **confrontando** due valori o confrontando più condizioni. Esempi di condizione:

`16 == 16`	`"a" == 'a'`	`12 + 3 >= 9 + 9`	`"Sì" == "sì"`	`età > 13 and età < 18`
`True`	`True`	`False`	`False`	`Dipende da variabile età`

L'istruzione `else`

L'istruzione **`else`**, che si traduce con *"altrimenti"*, è un'istruzione che viene **aggiunta** all'istruzione **`if`** e permette di eseguire un blocco di codice quando la condizione dell'**`if`** risulta **falsa**.
La sintassi è la seguente:

```
if condizione:
    <blocco di codice da eseguire se la condizione è True>
else:
    <blocco di codice da eseguire altrimenti>
```

Di seguito un esempio dell'utilizzo della struttura `if-else`:

```
numero = input("Inserisci un numero intero: ")
if numero >= 0:
    print("Il numero è positivo")
else:
    print("Il numero è negativo")
```

L'istruzione `elif`

L'istruzione `elif`, che sta per "else if", è un'istruzione che viene **aggiunta** all'istruzione `if` e permette di eseguire un blocco di codice quando la condizione dell'`if` risulta **falsa e si vuole verificare un'ulteriore condizione**. Deve essere aggiunta prima di un eventuale else, che non è obbligatorio.
La sintassi è la seguente:

```
1 if condizione:
2     <blocco di codice da eseguire se la condizione è True>
3 elif condizione2:
4     <blocco di codice da eseguire se la condizione2 è True>
5 else:
6     <blocco di codice da eseguire altrimenti>
```

Come già specificato, le righe 5 e 6 non sono obbligatorie.
Inoltre, si possono avere molteplici `elif` all'interno di una struttura `if`.

`if` annidati

È possibile **annidare** molteplici strutture `if` (`-elif-else`) all'interno di strutture già esistenti. Questa pratica è molto utile per **evitare di scrivere condizioni troppo lunghe e complesse**. Tuttavia, è bene utilizzarla con **parsimonia**, per evitare che il codice diventi una **scaletta** di `if`, risultando poco leggibile: è meglio non andare oltre i 5 `if` annidati.

```
if età >= 18 and registrato and accettato_termini:
    print("Accesso consentito.")
else:
    print("Accesso negato.")
```

```
if età >= 18:
    if registrato:
        if accettato_termini:
            print("Accesso consentito.")
        else:
            print("Devi accettare i termini.")
    else:
        print("Non sei registrato.")
else:
    print("Devi essere maggiorenne.")
```

Esercizi sugli `if` da svolgere insieme

① *Scrivere un programma che, a partire da un numero intero inserito dall'utente, dica se tale numero è pari o dispari.*

② *Scrivere un programma che, a partire da due numeri interi inseriti dall'utente, dica qual è il numero maggiore tra i due oppure se sono uguali.*

②b *Scrivere lo stesso programma, ma con tre numeri accettati dall'utente.*

③ *Scrivere un programma che chieda all'utente una stringa composta da un solo carattere. Se la stringa fornita dall'utente è più lunga di un carattere restituire un errore e terminare il programma. Altrimenti printare se la stringa è una vocale oppure no.*

④ *Scrivere un programma che chieda all'utente il suo reddito annuo e calcoli l'ammontare delle tasse dovute basandosi su semplici fasce di reddito:*

- *Redditi fino a 10.000€: esenti da tasse.*
- *Redditi superiori a 10.000€ e fino a 20.000€: tassati al 10%.*
- *Redditi superiori a 20.000€ e fino a 30.000€: tassati al 20%.*
- *Redditi superiori a 30.000€: tassati al 30%.*

Il programma deve mostrare l'ammontare delle tasse dovute secondo le suddette regole.

Esercizione

Creare un'applicazione bancaria che permette di effettuare un prelievo o un deposito monetario.

1. *Il programma deve permettere di effettuare il login chiedendo, separatamente, il nome utente e la password. Ci sono solo due utenti registrati (vedi sotto). Se il nome utente o la password sono errati (= la coppia non è corretta oppure il nome utente non esiste), printare un messaggio di errore e terminare il programma.*
2. *Se il login è corretto, stampare il bilancio attuale del conto e chiedere se l'operazione desiderata è "prelievo" o "deposito". Qualunque altro inserimento risulta nella terminazione del programma con un print di errore.*

Aiutino: `op = input("Prelievo (p) o deposito (d)? ") # L'utente quindi deve inserire la stringa "p" o "d", tutto il resto non è valido`

3. *Se l'operazione è di deposito, chiedere la cifra (si assume che l'utente non faccia lo stupido e inserisca sempre un numero intero), aggiornare il totale del conto e stamparlo a video, terminando poi il programma.*
4. *Se l'operazione è di prelievo, invece, chiedere la cifra (anche qui si assume che l'utente inserisca sempre un numero intero) e verificare che il bilancio sia sufficiente. Se lo è detrarlo dal totale e stampare il nuovo bilancio a video, terminando poi il programma. Se non lo è terminare il programma con un messaggio di errore.*

DATABASE:

```
user1 = "nick"              psw1 = "ildrugodrago12"       bilancio1 = 200
user2 = "biero"             psw2 = "FerrariPurosangue"    bilancio2 = 3000
user2 = "ElonioMuschio"     psw2 = "mars_emperor"         bilancio2 = 10000
```

Bonus: funzione input() e type casting

Le strutture di controllo

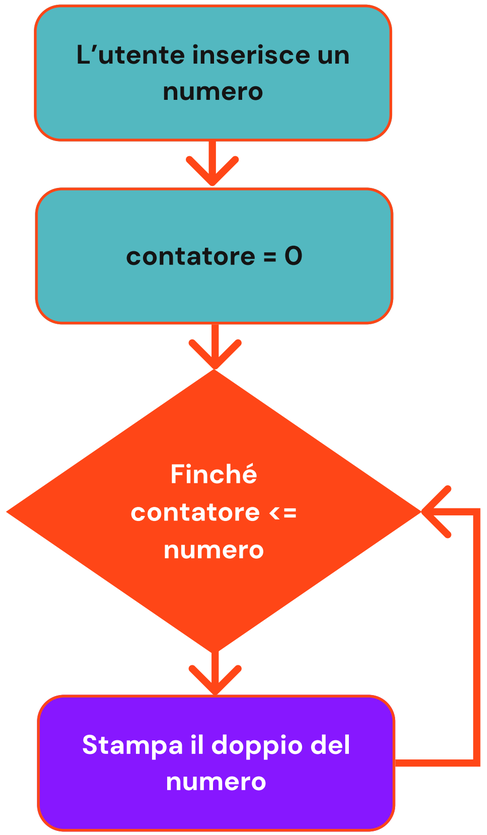
Condizionali (if, else, elif)

→ **Ciclo while**

I cicli

I **cicli** sono fondamentali per consentire al programma di *ripetere operazioni* (iterare) in modo efficiente. Dal punto di vista tecnico, i cicli eseguono un **blocco di codice ripetutamente** fintanto che una **condizione** specifica è soddisfatta (`True`). In sostanza, quando la condizione è `True`, il blocco di codice viene eseguito e la condizione viene nuovamente valutata. Se la condizione rimane `True`, il blocco di codice viene ripetuto, continuando così fino a quando la condizione risulta `True`.

Il **vantaggio** dell'utilizzo dei cicli è che permettono di **automatizzare e semplificare** compiti che richiederebbero ripetizioni di codice. In Python esistono due cicli: il ciclo `while` e il ciclo `for`. Il ciclo `while` si basa sulla valutazione di una **condizione booleana**. Il ciclo `for`, invece, è quello che in altri linguaggi di programmazione si chiama `for-each`. Infatti, non si basa sulla valutazione di una condizione booleana, ma sull'**esplorazione** di tutti gli **elementi** di una specifica **struttura dati**, ad esempio una **lista**. Per questo motivo, analizzeremo il ciclo `for` separatamente.



Il ciclo `while`

L'istruzione `while` verifica il **valore di verità** di una specifica **condizione** e, se tale valore risulta vero (`True`), allora esegue il **blocco di codice** corrispondente. Dopo tale esecuzione, la **condizione** viene nuovamente valutata e, se risulta di nuovo `True`, il blocco di codice viene **rieseguito**. Ciò succede **fintantoché** la condizione risulta `True`.
La sintassi è la seguente:

```
1 while condizione:
2     <blocco di codice da eseguire se la condizione è True>
```

Loop infinito

Il ciclo `while` può causare l'entrata in un **loop** (ciclo) **infinito**. Se la **condizione** risulta sempre `True` senza poter essere modificata, infatti, l'esecuzione del blocco di codice contenuto all'interno del ciclo si ripeterà all'infinito. Ciò può essere un problema, ma in alcune situazioni, invece, può essere necessario, come vedremo.

```
1 while True:
2     <blocco di codice eseguito indefinitamente>
```

Per terminare l'esecuzione di un programma in loop infinito, si deve utilizzare la combinazione di tasti Ctrl+C

Variabile contatore

La modalità in cui è stata introdotta la struttura del ciclo **`while`** potrebbe far pensare che venga utilizzata **solo** con **condizioni booleane** complesse e ben definite. Questo è certamente vero. Tuttavia, il ciclo **`while`** viene maggiormente impiegato quando si desidera **eseguire un blocco di codice un numero specifico di volte**. Per ottenere questo risultato, di solito si utilizzano delle variabili dedicate, comunemente chiamate **contatori**. Queste variabili sono **interi** inizializzati a `0` e il loro valore viene modificato all'interno del ciclo (*incrementato* o *decrementato*) fino a quando diventa *maggiore* o *minore* di un numero specifico. In questo modo, il ciclo viene eseguito un numero preciso di volte.

```
1 contatore = 0
2 while contatore < 10:
3     <blocco di codice da eseguire 10 volte>
4     contatore += 1
```

L'espressione `contatore < 10` rappresenta comunque una condizione booleana. Questa specifica sui contatori viene fornita per illustrare l'uso comune dei cicli `while`. Dal punto di vista sintattico, il funzionamento rimane invariato.

In questo modo, finché la variabile `contatore` sarà **minore di `10`**, il blocco di codice verrà eseguito **a ripetizione**. Allo stesso tempo, essendo `contatore` inizializzata a **`0`** e *incrementata* (+=) di `1` ad ogni **iterazione**, il blocco di codice verrà eseguito esattamente **`10`** volte.

Spesso le variabili contatori vengono chiamate `i`, `j` oppure `k`. È una semplice convenzione.

Istruzioni `break` e `continue`

Esistono due istruzioni che permettono di alterare la normale esecuzione dei cicli. Tali istruzioni si chiamano break e continue. Possono essere utili, ad esempio, quando si utilizzano i cicli per "cercare" qualcosa.

Istruzione `break`

L'istruzione `break` interrompe immediatamente l'esecuzione del ciclo e tutte le sue future iterazioni.

```
contatore = 1
while contatore <= 10:
    print(contatore)
    if contatore == 5:
        break
    contatore += 1
```

In questo esempio, utilizziamo un ciclo `while` per contare da **1** a **10**. Tuttavia, attraverso un `if` che verifica se il numero è **uguale a 5**, utilizziamo l'istruzione `break` per **interrompere** immediatamente l'**esecuzione del ciclo**. In questo modo, il programma stamperà i numeri da 1 a 4 e poi si fermerà.

Istruzione `continue`

L'istruzione `continue` interrompe immediatamente l'esecuzione dell'iterazione corrente, e salta a quella successiva.

```
contatore = 1
while contatore <= 10:
    if contatore % 2 == 0:
        contatore += 1
        continue
    print(contatore)
    contatore += 1
```

In questo esempio, utilizziamo un ciclo `while` per contare da **1** a **10**. Utilizziamo, però, l'istruzione `continue` per saltare l'iterazione quando il contatore è **pari**. In questo modo, il programma, quando entrerà nell'`if` che verifica se il contatore è pari, continuerà all'iterazione successiva, senza stampare. Quindi, il programma stampa tutti i numeri **dispari** da 1 a 10.

Esercizi sui `while`

1. *Scrivere un programma che esegue un conto alla rovescia partendo da un numero intero fornito in input dall'utente e arrivando fino a 0.*
2. *Scrivere un sistema di verifica della password. Il programma deve chiedere all'utente una password e, se la password è corretta (password = "Pythonata2024"), il programma deve stampare "Accesso consentito", altrimenti, la password deve essere inserita e verificata nuovamente.*
3. Scrivere un programma che simula un processo di controllo qualità in una linea di produzione industriale. Il programma deve chiedere all'utente di inserire il risultato di un controllo qualità, che può essere "passato" o "fallito". Il programma continua a chiedere i risultati finché non viene inserito "fallito". Quando un controllo fallisce, stampare "Prodotto difettoso trovato, linea di produzione fermata." e terminare il programma. Qualora l'utente dovesse inserire la stringa "esci", invece, terminare immediatamente il programma senza ulteriori messaggi.
4. *Scrivere un programma che continua a chiedere all'utente di inserire numeri interi. Il programma deve ignorare (non elaborare) i numeri negativi inseriti, mentre deve stampare il quadrato di tutti i numeri positivi inseriti. In altre parole, se il numero inserito è negativo, il programma non deve fare nulla e chiedere il prossimo numero. Se, invece, è positivo, il programma deve stampare il suo quadrato. Se l'utente inserisce il numero 0, invece, il programma deve terminare.*

Grazieng!