UNIVERSITÀ di VERONA

Linguaggio di Programmazione Python

04/04/2022

Giulio Mazzi

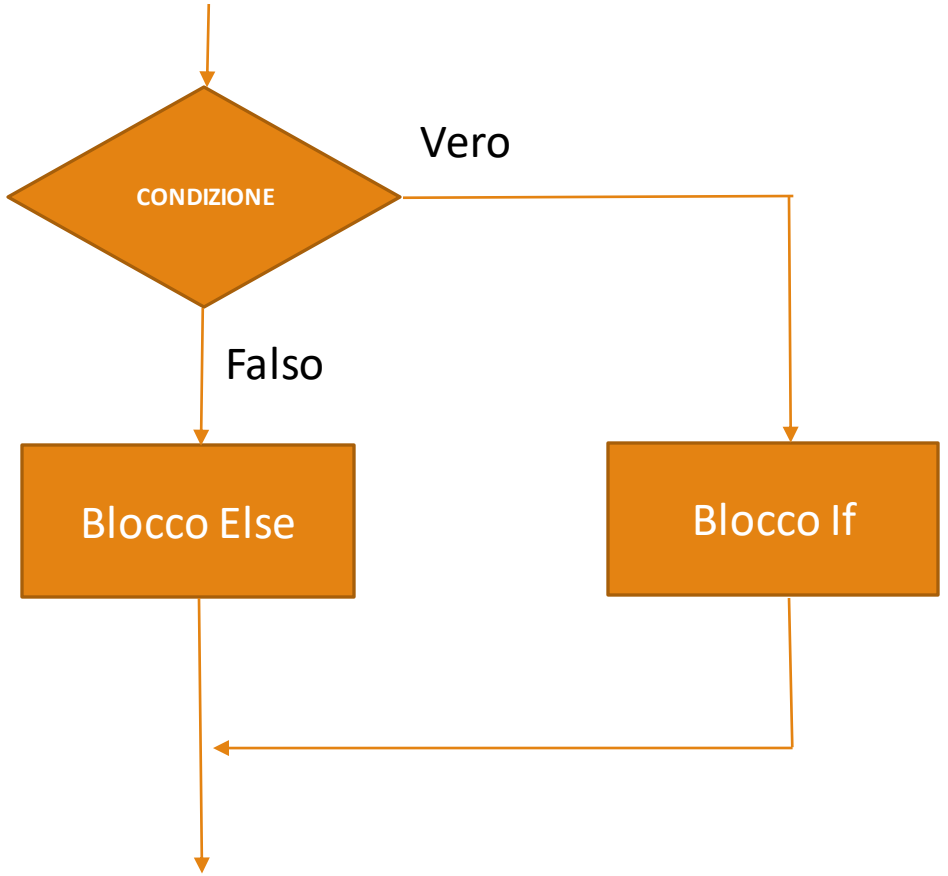
giulio.mazzi@univr.it

Istruzioni condizionali

Fino a ora, i nostri programmi si sono comportati come "ricette", ovvero liste d'istruzioni da eseguire una alla volta. In informatica queste spesso vengono chiamate *procedure*

Tuttavia, siamo interessati a programmi che **prendono decisioni** usando i dati in loro possesso.

L'istruzione **if** viene utilizzata per verificare condizione, se è vera viene eseguito un blocco (il **blocco if**) altrimenti un altro blocco (il **blocco else**)



Istruzioni condizionali

- L'istruzione if viene utilizzata per verificare condizione, se è vera viene eseguito un blocco (il blocco if) altrimenti un altro blocco (il blocco else)
- La clausola else è opzionale
- In python delimitiamo la condizione con **:**, blocco if e blocco else sono allo stesso livello d'indentazione

```
if voto >= 18:
    print('complimenti!')
    print('promosso')
else:
    print('bocciato')
```

```
if x < 0:
    x = -x
print(x)
```

Condizione

- Le condizioni sono implementate come delle espressioni
  - In genere ritornano vero (**True**) o falso (**False**), notare le maiuscole
  - Anche altre istruzioni possono essere valutate in questo modo (e.g., numeri, stringhe...)

Operatore	Significato
==	Uguale (semanticamente) a
!=	Diverso da
<	Minore di
>	Maggiore di
<=	Minore o uguale a
>=	Maggiore o uguale a
is	Uguale (in memoria) a

Combinare

Posso concatenare tra loro più espressioni Booleane:

- L'istruzione **and** è vera solo se entrambi I suoi membri sono veri
- L'istruzione **or** è vera se uno dei suoi membri è vero (or inclusivo)

Con l'operatore **not** posso negare una variabile Booleana

N.B. le operazioni aritmetiche hanno la precedenza su quelle di confronto. Le operazioni di confronto hanno la precedenza sugli operatori logici

```
x = 3
editor = 'vim'

if x > 0 and x <= 10:
    print('x è tra 1 e 10')

if x == 1 or x == 3:
    print('x è 1 o 3')

if not editor == 'vim':
    print('errore')
```

If inline (o ternario)

L'operazione d'inizializzare una variabile a diversi valori usando un if è molto comune, python ci semplifica la vita.

Pensate alla frase:

"abs_x è x se x è maggiore o uguale a 0, altrimenti è - x"

Notare che non si usano ":"

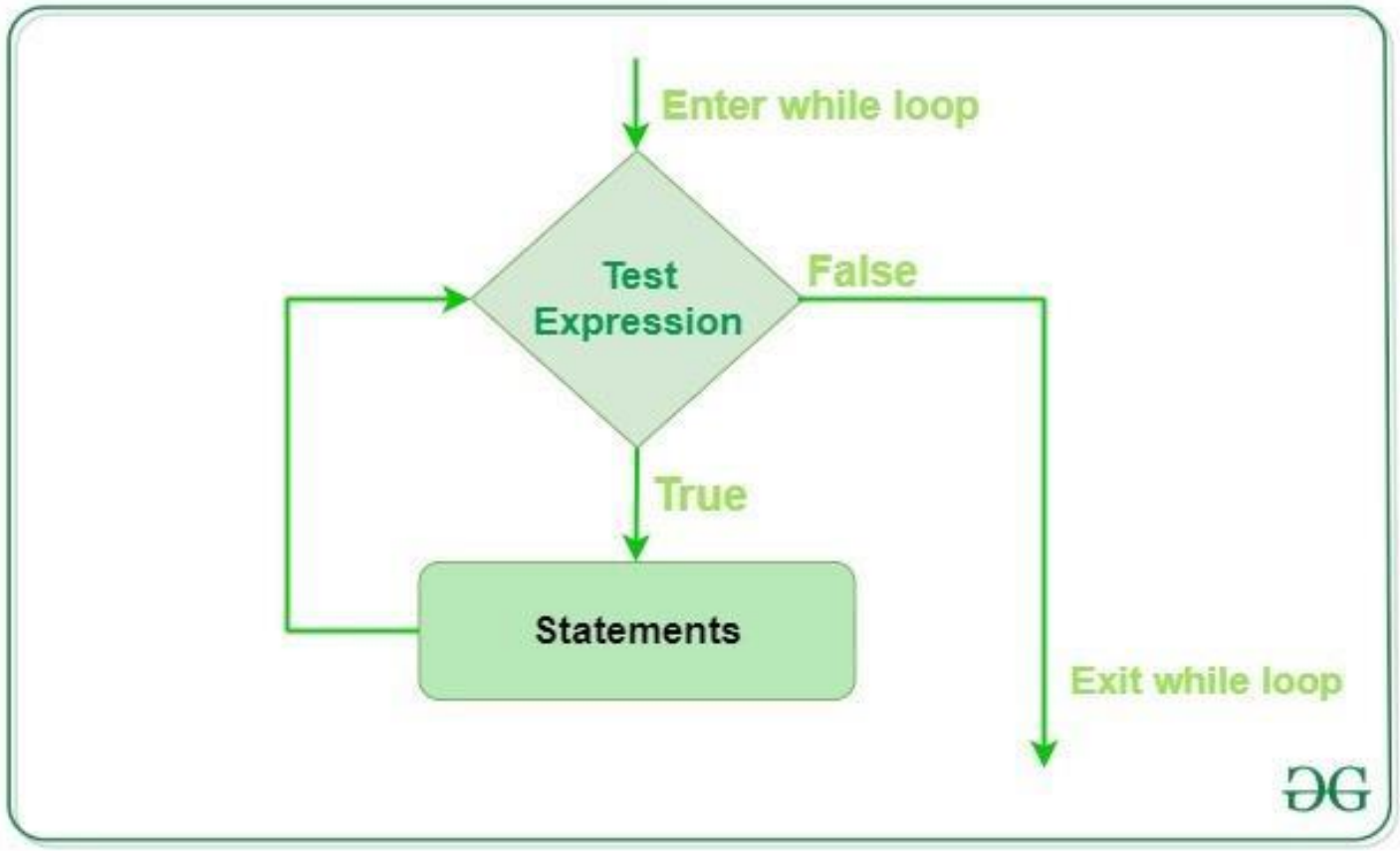
```
x = -10

# versione estesa
abs_x = x
if abs_x < 0:
    abs_x = -abs_x
print(x, abs_x)

# inline if, più compatto!
abs_x = x if x >= 0 else -x
print(x, abs_x)
```

Il ciclo while

Con l'if, il nostro codice può prendendere decisioni... ma si può fare di meglio!

Il while ci permette di ripetere più volte un blocco d'istruzione più volte.

A ogni *ciclo* testiamo il valore di un espressione e ci fermiamo solo quando diventa falsa



While in python

La struttura sintattica del **while** è molto simile a quella dell'**if**

Notate che anche qui usiamo : per identificare la condizione

```
y = 0
while y < 10:
    print(y)
    y = y + 1
```

Cosa succede se la condizione non è mai vera?

Break & continue

- Possiamo usare **continue** per passare "al prossimo giro" (ovvero tornare alla condizione)

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

- Possiamo usare **break** per saltare subito fuori da un ciclo

```
y = 0
while y < 10:
    if y == 5:
        print('trovato')
        break
    y = y + 1
    print(y)
```

While… else?

In python è possibile definire l'else di un ciclo for, serve a semplificare un *pattern* molto comune.

Il codice nel ramo else è eseguito solo se non uso mai il break

```
piatti = ['pasta', 'pizza', 'risotto']
i = 0
while i < len(piatti):
    if piatti[i] == 'insalata':
        print('un insalata per favore')
        break
    else:
        i += 1
else:
    print("non c'è insalata")
```

Cicli for

Il ciclo **while** non è l'unico modo di ripete più volte un blocco di istruzioni. Possiamo usare anche il **for**. In python, il for itera su una serie di elementi, in altri linguaggi questo viene chiamato **for each.**

```
gioco = [1, 2, 3, 'stella']
for fase in gioco:
    print(fase)
```

Posso usarlo per semplificare il codice, non mi serve lavorare con gli indici

```
piatti = ['pasta', 'pizza', 'risotto']
for p in piatti:
    if p == 'insalata':
        print('un insalata per favore')
        break
else:
    print("non c'è insalata")
```

Varianti del for

Esistono diversi modi di usare il ciclo for

- Per iterare su una lista di numeri posso usare **for n in range(10):**, che ritorna i numeri da 0 a 9
- Per iterare contando le posizioni posso usare **for pos, val in enumerate(lista):**
- Per iterare su più liste in contemporanea posso usare **for n, m in zip(nomi, matricole):**

Vediamo qualche esempio in jupyter...

Tuple

Alcune delle istruzioni usate precedentemente (zip, enumerate) costruiscono delle tuple.

In Python, le tuple, come le stringhe, sono sequenze immutabili.

Si usano per gestire collezioni di oggetti che non cambiano nel tempo

- t = (1,2,'Ciao',4)
- t2 = tuple(…) dichiarazione esplicita
- t3 = (1,) notare che le tuple con un solo elemento richiedono la virgola

N.B. gli elementi all'interno possono essere mutabili!

Manipolare le tuple

```
piatti = ['pasta', 'pizza', 'risotto']
tavolo = ('sedie', 'tovaglia', piatti)
print(piatti, tavolo)

# OK: piatti è una lista mutabile
tavolo[2].append('insalata')

# OK: le modifiche sono viste all'interno di tavolo
piatti.append('gelato')

# ERRORE! non posso aggiungere alla tupla
tavolo.append('bicchieri')

# ERRORE: non posso assegnare, la stringa è immutabile
tavolo[0] = 'divani'
```

Unpacking

Le tuple sono estremamente utili per organizzare i dati. A volte però è comodo spezzarli in variabili singole, facile da fare in python:

```
numeri = (1, 2)
x, y = numeri
print(x, y) # stampa 1 2
```

Questo è utile per gestire funzioni che ritornano più di un valore. Inoltre possiamo usarle per fare uno scambio (detto swap) tra variabili.

```
def calcolo(x):
    # ... calcola qualcosa ...
    return 10, 0.5

valore, errore = calcolo(100)
```

```
# swap
x = 10
y = 5
x, y = y, x

# stampa 5, 10
print(x, y)
```

Tuple - riassumendo

immutabile, contiene elementi eterogenei.

Tupla:	(elemento, elemento, elemento, ...)
Tupla vuota	()
Tupla 1 elemento	(elemento,)
tipo	tuple
creatore	tuple(...)

Funzioni e metodi e principali

tupla[...]	estrae elementi (non li cancella, ovviamente).
len(tupla)	lunghezza della lista
x **in** tupla	True se l'elemento x è nella tupla

Essendo immutabile non ci sono metodi per eliminare/aggiungere elementi. Attenzione che come detto precedentemente ciò che si memorizza sono sempre riferimenti. Se l'elemento è mutabile lo posso modificare, ma non posso far puntare quella particolare locazione della tupla a un altro elemento.