Loops e condizioni

October 25, 2020

1 Loops e condizioni

Nel corso del nostro programma potrebbe essere richiesto di porre delle condizioni a certe variabili o istruzioni o istruzioni che dovranno essere ripetuti e quindi necessitano di loops, python fornisce i seguenti strumenti a tal caso: - Condizioni - if - elif - else - Loops - while - for - list comprehesion

1.1 Condizioni

Le istruzioni **if, else, elif** servono per eseguire blocchi di codici qualora delle specifiche condizioni fossero incontrare. ### if ###

[1]:
```
#definiamo una variabile x
x = 5
#creiamo delle condizioni da applicare
if x > 0:
    #il blocco di if deve essere indentato
    #tutto il codice indentato verrà interpretato
    print(x,'è positivo')
```

```
5 è positivo
```

Vediamo di capire come viene interpretato il codice, il comando `if` viene seguito da una condizione che verrà valutata e convertita in **booleano**(vero o falso) qualora la condizione sia vera, verrà eseguito il blocco di codice contenuto indentato.

[2]:
```
#la condizione viene convertita in booleano
x > 0
```

[2]:
```
True
```

1.1.1 else, elif

Ora qualora noi vorremmo valutare ulteriori condizioni è possibile usare `else` o `elif`(else if), `elif` viene usato per aggiungere ulteriori condizioni ulteriori a `if` e viene valutata subito dopo di essa, `else` invece viene valutata qualora le condizioni precedenti siano risultati false.

[3]:
```
if x < 0:
    #if viene valtutato per primo
    print(x,'è negativo')
```

```
elif x > 0:
    #il secondo che viene valutato è if else
    print(x,'è positivo')
else:
    #else viene valutato per ultimo
    print(x,'è nullo')

x = 0
if x < 0: #falso
    print(x,'è negativo')
elif x > 0: #falso
    print(x,'è positivo')
else: #vero, poichè le altre condizioni sono false
    print(x,'è nullo')
```

```
5 è positivo
0 è nullo
```

```
Le condizioni <b>else</b> e <b>elif</b> non sono obbligatorie, ma è buona norma usarle per capi
```

Infatti qualora noi non usassimo else e le condizioni precedenti fossero false ecco quello che succederebbe:

[4]:
```
if x < 0: #falso
    print(x,'è negativo')
elif x > 0: #falso
    print(x,'è positivo')
```

Non è presente alcun risultato ed il motivo è semplice, le condizioni precedenti sono false e pertanto non viene eseguito il blocco di codice contenuto al loro interno.

1.1.2 Abbreviazione di un semplice if else

In molti casi dovremmo utilizzare una semplice condizione if seguita solo da un else, in tal caso è possibile usare un **operatore ternario** che permette di definire la condizione precedente in una singola riga, vediamolo:

[5]:
```
x = 1
#la forma della condizione è la seguente
#in questo caso la condizione è vera poiché x > 0
print('positivo') if (x > 0) else print('negativo o nullo')
#cambiando il valore di x e facendo diventare la condizione falsa
x = -1
print('positivo') if (x > 0) else print('negativo o nullo')
```

```
positivo
negativo o nullo
```

Analizziamo come è composta la condizione precedente: `condizione è vera if condizione else condizione è falsa` quindi se la condizione sarà vera verrà eseguito il codice prima di

`if` altrimenti verrà eseguito il codice dopo `else`, la condizione è posta tra di essi.

1.1.3 Condizioni multiple

In python è possibile valutare condizione multiple con queste istruzioni utilizzando gli **operatori bitwise** `and`, `or`, `xor`, ce ne sono altri come `compliment`,`left shift` e `right shift`, ma non li tratteremo poiché non le useremo se siete comunque curiosi potete consultare questo **link**.

and L'operatore bitwise `and` o `&` è il classico **operatore logico AND** tale per cui il risultato è vero solo se entrambe le condizioni sono vere, questa tabella mostra la logica:

Condizione1	Condizione2	Risultato
True	True	True
True	False	False
False	True	False
False	False	False

[6]:
```
#proviamo ad applicare questo ad un if
#stesso modo è possibile applicarlo ad elif
x = 10
if (x > 0) and (x < 20): #entrembi vere
    print(x,'è compreso tra 0 e 20')
#sempre and, ma con carattere &
if (x > 0) & (x < 20): #entrambi vere, sempre and
    print('& e and hanno lo stesso comportamento')

if (x > 0) and (x < 10): #una delle due è false
    #questo blocco non verrà interpretato
    print(x,'è compreso tra 0 e 10')
```

```
10 è compreso tra 0 e 20
& e and hanno lo stesso comportamento
```

or L'operatore bitwise `or` o `|` è il classico **operatore logico OR** tale per cui il risultato è sempre vero, tranne nel caso in cui entrambi siano falsi, questa tabella mostra la logica:

Condizione1	Condizione2	Risultato
True	True	True
True	False	True
False	True	True
False	False	False

[7]:
```
#proviamo ad applicare questo ad un if
#stesso modo è possibile applicarlo ad elif
x = 10
```

```
if (x > 0) or (x < 0) : #una delle due è vera
    print(x,'è positivo o negativo')
#sempre or, ma con carattere |
#potrebbe dover richiedere parentesi
if (x > 0) | (x < 0) :
    print('or e | sono equivalenti')

#solo se entrambe sono false, non verrà valutato
if (x < 10) or (x < 0): #entrambe sono false
    #questo blocco non verrà interpretato
    print(x,'è negativo o minero di 10')
```

```
10 è positivo o negativo
or e | sono equivalenti
```

xor L'operatore bitwise xor è definito con ^ ed il risultato è vero solo se una delle due condizioni è falsa, altrimenti il risulato sarà falso:

Condizione1	Condizione2	Risultato
True	True	False
True	False	True
False	True	True
False	False	False

[8]:
```
#proviamo ad applicare questo ad un if
#stesso modo è possibile applicarlo ad elif
x = 10
if (x >= 10) ^ (x < 0): #una delle due è falsa
    print(x,'è maggiore uguale a 10 o minore di 0')
if (x < 0) ^ (x < 10): #entrambe sono false
    #questo blocco, non verrà interpretato
    print(x,'è negativo o minore di 10')
if (x > 0) ^ (x > -10): #entrambe sono vere
    #questo blocco, non verrà interpretato
    print(x,'è positivo o compreso tra -9 e 0')
```

```
10 è maggiore uguale a 10 o minore di 0
```

```
Ricordate che in alcuni casi potrebbe non essere valutata l'operatore poiché la condizione <b>
```

```
E' inoltre possibile uniore più di due condizioni usando sempre gli operatori bitwise.
```

1.2 Loops

I loops sono usati per eseguire un numero di istruzioni N volte dove N è un valore specificato da noi, vediamo le loro appicazioni

1.2.1 while

Il comando `while` viene usato per ripetere un blocco di codice fino a che la condizione imposta non risulta più vera.

[9]:
```
i = 0 #definiamo una variabile globale
while (i < 5):
    #fino a che la condizione precedente
    #risulta vera, esegui questo blocco
    print('Il valore i è:',i)
    #ricordate di aggiornare la variabile
    #usata per la condizione altrimenti
    #non uscirete dal loop!
    i += 1 # i = i + 1 incrementa di 1
```

```
Il valore i è: 0
Il valore i è: 1
Il valore i è: 2
Il valore i è: 3
Il valore i è: 4
```

```
Notate bene che quando `i = 5` il loop esce <b>senza eseguire il blocco contenuto</b>.
```

break Qualora la condizione sia **sembre vera** è possibile usare il comando `break` per uscire dal loop.

[10]:
```
i = 0 #definiamo una variabile
while True: #in questo caso la condizione è sempre vera
    if i >= 5: #la condizione se vera, ci farà uscire dal loop
        break  #questo comando ci fa uscire dal loop
    print('Il valore i è:',i)
    #ricordate di aggiornare i altrimenti l'if
    #non sarà mai vero!
    i += 1 # i = i +1
```

```
Il valore i è: 0
Il valore i è: 1
Il valore i è: 2
Il valore i è: 3
Il valore i è: 4
```

continue Qualora invece volessimo che il loop **continui senza però interpretare il codice nel blocco**, in tal caso il comando `continue` applica questo.

[11]:
```
i = 0
while i <= 10:
    if i % 2 == 0: #se il numero è pari
        i += 1 #ticordate di incrementare i
        #questo comando dirà di non intepretare
```

```
        #il codice di blocco successivo contenuto
        # nel while
        continue
    #eseguito solo se non viene intepretato
    #il comando continue
    print('Il valore i è:',i)
    i += 1
```

```
Il valore i è: 1
Il valore i è: 3
Il valore i è: 5
Il valore i è: 7
Il valore i è: 9
```

1.2.2 For

Il comando `for` è leggermente diverso dal `while` in python, poiché in questo caso noi definiamo su quali elementi sia necessario loopare, senza specificare la condizione da applicare, per essere più chiari introduciamo prima il comando `range`. #### range #### Il comando `range` crea una sequenza di valori partendo da un valore iniziale `start` a un valore finale `end`, poiché range appartiene alla classe 'range' in genere è solito convertirlo in lista per mostrare la sua intera sequenza.

[12]:
```
print(range(10)) #questo ci darà solo valore finale e iniziale
```

```
range(0, 10)
```

[13]:
```
print(list(range(10))) #questo ci darà tutti gli elementi contenuti
print(list(range(2,10))) #lista da 2 a 9, non 10!
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7, 8, 9]
```

Notare bene che qualora non sia specificato lo `start` il valore di default è 0 e la sequenza contiene tutti gli elementi fino al valore precedente di `end`.

Utilizzo di for e range Il comando `range` è molto utilizzato insieme ad `for`, vediamo alcuni esempi.

[14]:
```
for j in range(10):
    #ricordate che j è una variabile locale
    #e non accessibile al di fuori del for se
    #prima non convertita a global
    print('il valore j è',j)
```

```
il valore j è 0
il valore j è 1
il valore j è 2
il valore j è 3
il valore j è 4
```

```
il valore j è 5
il valore j è 6
il valore j è 7
il valore j è 8
il valore j è 9
```

Notate molto attentamente che qui **non è stata usata una condizione, ma il comando `in` per specificare dove loopare il nostro indice**, in questo caso una sequenza di numeri da 0 a 9.

Utilizzo for in liste e dizionari

[15]:
```
#il ciclo for e il comando in è molto utilizzato sulle liste, dizionari, set e␣
 ↪molto altro
lista = ['ciao', 4, 17, 'tredici']
for j in lista:
    #ora j diventerà ad ogni loop
    #un singolo elemento della lista
    print('lista contiene:',j)
```

```
lista contiene: ciao
lista contiene: 4
lista contiene: 17
lista contiene: tredici
```

[16]:
```
#può essere anche usato per accedere le liste attraverso l'indice
#len(lista) fornisce il numero di elementi dentro lista
for j in range(len(lista)):
    #in questo caso useremo j per accedere attraverso l'indice la lista
    print('lista in posizione',j,'contiene',lista[j])
```

```
lista in posizione 0 contiene ciao
lista in posizione 1 contiene 4
lista in posizione 2 contiene 17
lista in posizione 3 contiene tredici
```

L'esempio precedente può essere anche definito usando `enumerate` che associa ad ogni elemento della lista un iterabile.

[17]:
```
print('lista:',lista)
#enumerate associerà un iterabile
enum_lista = enumerate(lista)
print('lista enumerata:')
for enum_elem in enum_lista:
    #ricordate di usarlo in un for altrimenti
    #verrà restituito un indirizzo
    print(enum_elem)
```

```
lista: ['ciao', 4, 17, 'tredici']
lista enumerata:
(0, 'ciao')
```

```
(1, 4)
(2, 17)
(3, 'tredici')
```

[18]:
```
for count, elem in enumerate(lista):
    #count verrà associato a 0,1,2,3
    #elem verrà associato a ciao,4,7,tredici
    print('la lista in posizione',count,'contiene',elem)
```

```
la lista in posizione 0 contiene ciao
la lista in posizione 1 contiene 4
la lista in posizione 2 contiene 17
la lista in posizione 3 contiene tredici
```

[19]:
```
capitali = {'Piemonte':'Torino', 'Lombardia':'Milano', 'Lazio':'Roma'}
for regione, capitale in capitali.items():
    #il comando items permette di accedere sia alle key che alle values
    print('la regione',regione,'ha capitale',capitale)
```

```
la regione Piemonte ha capitale Torino
la regione Lombardia ha capitale Milano
la regione Lazio ha capitale Roma
```

[20]:
```
for regione in capitali: #considererà solo le keys
    print(regione)
```

```
Piemonte
Lombardia
Lazio
```

[21]:
```
for capitale in capitali.values(): #considererà solo le values
    print(capitale)
```

```
Torino
Milano
Roma
```

[22]:
```
for regione in capitali:
    #altro modo per accedere alle values
    print(capitali[regione])
```

```
Torino
Milano
Roma
```

1.2.3 Comprensioni di liste

In genere se si vogliono utilizzare dei loop per applicare delle operazioni su una lista è possibile usare la **comprehension list** che permette di loopare dentro la lista attraverso una singola riga.

[23]:
```
numbers = list(range(10)) #lista da 0 a 9
print('numbers prima della conversione in booleana:',numbers)
#faccio diventare numbers in una lista di true false con una condizione
bools = [x == 0 for x in numbers] #true solo se l'elemento è 0
print('numbers convertita in zeri con la compresione di liste:',bools)
```

```
numbers prima della conversione in booleana: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
numbers convertita in zeri con la compresione di liste: [True, False, False,
False, False, False, False, False, False, False]
```

La compresione di lista è quindi nella seguente forma `[espressione for iterabile in lista]` l'iterabile diventa quindi un elemento della lista e nella espressione è possibile porre una qualsiasi condizione.

[24]:
```
#altro esempio sulla lista di comprensione
num_parity = ['pari' if (x % 2)== 0 else 'dispari' for x in numbers]
print(num_parity)
```

```
['pari', 'dispari', 'pari', 'dispari', 'pari', 'dispari', 'pari', 'dispari',
'pari', 'dispari']
```

[25]:
```
#l'esempio precedente con il classico loop e if sarebbe stato
num_parity = numbers #copia della lista numbers originale
for x in range(len(num_parity)):
    #valuto l'elemento all'indice x
    if (num_parity[x] % 2) == 0: #potevo mettere il blocco anche qui, se␣
 ↪composto da una sola riga
        #ridefinisco l'elemento in pari
        num_parity[x] = 'pari'
    else:
        #altrimenti lo ridefinisco in dispari
        num_parity[x] = 'dispari'
print(num_parity)
```

```
['pari', 'dispari', 'pari', 'dispari', 'pari', 'dispari', 'pari', 'dispari',
'pari', 'dispari']
```

COMPLIMENTI AVETE RAGGIUNTO LA FINE DELLA LEZIONE SU LOOPS E CONDIZIONI!