ciclo-for

June 5, 2023

0.1 Cicli for

Con i costrutti `if elif else` e le funzioni abbiamo visto che il *flusso del programma* può evolvere in maniera non necessariamente lineare. Il ciclo `for` permette di **ripetere** numerose volte la stessa porzione di codice: - ogni ripetizione produce un piccolo progresso; - l'esecuzione complessiva produce un grande effetto.

Come sempre, prima di discutere sintassi e funzionamento esatto, partiamo con un esempio concreto.

[1]:
```
lista = [1,5,78,3,6]

for x in lista:
    print(x)
```

```
1
5
78
3
6
```

Il ciclo ripete l'istruzione `print(x)` con `x` che assume uno alla volta il valore degli elementi in `lista`.

La sua sintassi è la seguente:

```
for <var> in <seq>:
    <istruzione1>
    <istruzione2>
    <istruzione3>
    ...
```

`<var>` è il nome di una variabile (non necessariamente esistente) che assumerà uno ad uno il valore degli elementi di , che può essere una qualunque espressione che ha come valore una sequenza. Ad ogni ripetizione viene eseguito il blocco di codice sottostante il comando `for`. La riga del comando `for` termina con i due punti, e quindi le righe successive devono formare un blocco di codice indentato verso destra. Se la lista contiene `N` elementi, il codice contenuto all'interno del ciclo viene eseguito `N` volte.

Vediamo ancora una semplice stampa di tutti gli elementi della sequenza.

[2]:
```
seq = [1, 15, 2+5, "casa"+"gatto", -6]

for x in (1, 15, 2+5) + ("casa"+"gatto", -6):
```

```
    print(x)
```

```
1
15
7
casagatto
-6
```

0.1.1 Esempi d'uso del ciclo `for`

Non c'è molto da dire su come si usa il ciclo `for`. La sintassi dovrebbe essere chiara, così come il suo funzionamento. Tuttavia quello che serve ora sono diversi esempi per comprenderne l'uso e le possibilità. Naturalmente questi esempi sono - elementari, - non coprono tutte le infinite possibilità d'uso, - possono essere combinati assieme.

Trasformazione: tutto maiuscolo

[3]:
```
sorgente = ["cane", "gatto", "canarino", "gallo"]
destinazione = []

for x in sorgente:
    destinazione.append( x.upper() )

print(destinazione)
```

```
['CANE', 'GATTO', 'CANARINO', 'GALLO']
```

Questo esempio è tipico: data una lista, vogliamo produrne un'altra della stessa lunghezza dove gli elementi della nuova lista sono una **trasformazione** dei corrispondenti elementi della lista vecchia. Per far questo: 1. creiamo un lista vuota, 2. per ogni elemento della lista vecchia aggiungiamo in coda alla nuova lista l'elemento trasformato.

Certo, adesso potremmo definire una funzione che fa la stessa cosa: - il ciclo `for` sarà all'interno della funzione; - la lista vuota viene creata e popolata all'interno della funzione; - la lista popolata viene restituita al programma chiamante; - vedete che ad ogni chiamata di funzione viene prodotta una **nuova** lista.

[9]:
```
def tuttomaiuscolo(seq):
    risultato = []
    for x in seq:
        risultato.append( x.upper() )
    return risultato
```

[10]:
```
sorgente1 = ["cane", "gatto", "canarino", "gallo"]
sorgente2 = ["nave", "automobile", "moto", "aereo"]
sorgente3 = ["storia", "geografia", "educazione civica", "lettere"]

destinazione1 =  tuttomaiuscolo(sorgente1)
destinazione2 =  tuttomaiuscolo(sorgente2)
destinazione3 =  tuttomaiuscolo(sorgente3)
```

```
print( destinazione1  )
print( destinazione2  )
print( destinazione3  )
```

```
['CANE', 'GATTO', 'CANARINO', 'GALLO']
['NAVE', 'AUTOMOBILE', 'MOTO', 'AEREO']
['STORIA', 'GEOGRAFIA', 'EDUCAZIONE CIVICA', 'LETTERE']
```

Esercizio: scrivete una funzione che prenda come parametro una lista di strighe, e restituisca una lista con le rispettive lunghezze.

Trasformazione: numeri al cubo

[11]:
```
def numerialcubo(numeri):
    cubi = []
    for numero in numeri:
        cubi.append( numero**3)
    return cubi
```

[13]:
```
L1 = [1, 2, 3, 4, 5]
print( numerialcubo(L1) )
```

```
[1, 8, 27, 64, 125]
```

[14]:
```
L2 = [ -4, 12, 4.5, 3, 0.5]
print( numerialcubo(L2) )
```

```
[-64, 1728, 91.125, 27, 0.125]
```

Come vedete lo stile del programma è simile a quello del programma precedente. E infatti il paradigma è lo stesso: elaboriamo uno ad uno gli elementi di una lista (o sequenza) in ingresso ed emettiamo il risultato su una lista in uscita.

Esercizio: scrivete una funzione che prenda come parametro una lista di numeri positivi, e restituisca una lista contentente stringhe costituite da sole `'a'`, di lunghezza pari al numero nella rispettiva posizione. - Esempio: con parametro `[3,5,2]` deve produrre `["aaa", "aaaaa", "aa"]` - Suggerimento: ricordate la moltiplicazione tra un numero e una stringa?

Statistiche : conteggio Sappiamo che la funzione `len` ci dice quanti elementi ci sono in una sequenza. Proviamo a scrivere noi una funzione che si comporta allo stesso modo. L'idea è - inizializzre un **contatore** a 0, - e per ogni elemento della lista **incrementare il contatore** di 1.

[15]:
```
def lunghezza(seq):
    cnt = 0
    for x in seq:
        cnt = cnt + 1
    return cnt
```

[16]:
```
I1 = ["cane", "gatto", "canarino", "gallo"]
I2 = ["nave", "automobile", "aereo"]
I3 = ["storia", "geografia", "educazione civica", "lettere", "ginnastica"]
I4 = [1, 2, 3, 4, 5, 6, 7, 8]
I5 = [ -4, 12, 4.5, 3, 0.5]

print( lunghezza(I1) )
print( lunghezza(I2) )
print( lunghezza(I3) )
print( lunghezza(I4) )
print( lunghezza(I5) )
```

```
4
3
5
8
5
```

Statistiche: somma totale Data una lista di numeri, Python ha la funzione predefinita `sum` che produce la somma di questi numeri. Proviamo a scrivere noi una funzione che si comporta allo stesso modo. L'idea è simile a quella dell'esempio precedente: - inizializzare un **accumulatore** a 0; - sommare all'accumulatore ogni elemento della lista.

[19]:
```
def sommatotale(seq):
    somma = 0
    for x in seq:
        somma = somma + x
    return somma
```

[20]:
```
I1 = [1, 2, 3, 4, 5, 6, 7, 8]
print( sommatotale(I1) )
```

```
36
```

[21]:
```
I2 = [ -4, 12, 4.5, 3, 0.5]
print( sommatotale(I2) )
```

```
16.0
```

Esercizio: invece della somma, calcolate la media aritmetica.

Filtraggio: solo i valori positivi Adesso vediamo un altro paradigma di elaborazione di sequenze: **selezioniamo** da una lista **solo gli elementi** che rispettano certe condizioni. Ad esempio data una lista di numeri filtriamo quelli che non sono strettamente maggiori di 0.

L'idea è simile alla trasformazione di una lista. Tuttavia ad ogni ripetizione l'elemento viene aggiunto alla lista solo se rispetta la proprietà desiderata.

[23]:
```
def filtrapositivi(seq):
    res = []
    for x in seq:
        if x > 0:
            res.append(x)
    return res
```

[24]:
```
I1 = [1, -2, -3, 4, -5, 6, 7, 8]
print( filtrapositivi(I1) )
```

```
[1, 4, 6, 7, 8]
```

[25]:
```
I2 = [ -4, 12, -4.5, 3, 0.5]
print( filtrapositivi(I2) )
```

```
[12, 3, 0.5]
```

Esercizio: quando si eseguono analisi di dati, spesso è necessario filtrare e pulire i dati da osservazioni errate, o valori mancanti, o sporchi. Combinate l'esempio del filtraggio e della somma totale e scrivere le seguenti funzioni. 1. Una funzione che data una lista di stringhe, conta quante sono quelle di lunghezza maggiore di 5. 1. Come la funzione precedente, ma la lunghezza minima adesso deve essere un parametro.

Esempio: con parametri `["casa", "cervo", "bue"]` e 4, deve restituire 2.

Esempio: con parametri `["casa", "cervo", "bue"]` e 5, deve restituire 1.

1. Uuna funzione che data una lista di numeri, calcoli la somma dei soli positivi.

Generatore di valori: i primi `n` quadrati In questo caso l'input **non è una sequenza**. Vogliamo generare una lista contentente i primi `n` numeri quadrati. In un certo senso questo è simile al paradigma in cui si trasforma la lista `[1,...,n]` applicando il quadrato ad ogni suo elemento. - Problema: non abbiamo la lista `[1,...,n]` come input, ma solo il numero `n`. - Soluzione: usiamo `range`!

[26]:
```
def primiquadrati(n):
    out = []
    for i in range(1,n+1):
       out.append(i**2)
    return out
```

[27]:
```
print( primiquadrati(10) )
print( primiquadrati(4) )
print( primiquadrati(0) )
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[1, 4, 9, 16]
[]
```

Esercizio: scrivete una funzione con un singolo parametro `n` che generi la lista dei numeri divisibili per 5.

0.1.2 Due modi di ciclare su una lista

Vediamo un modo alternativo di ciclare su una sequenza. Invece di ciclare con `for x in seq:` e poi usare `x` per riferirci agli elementi, possiamo osservare che - si può accedete all'elemento i-esimo di `seq` usando `seq[i]` - le posizioni accessibili di `seq` vanno da `0` a `len(seq)-1` incluso.

[28]:
```
seq = ('miao', 32, 'casa', 'gatto', -7, 1.2)
```

[29]:
```
# cicliamo per elemento
for x in seq:
    print(x)
```

```
miao
32
casa
gatto
-7
1.2
```

[30]:
```
# range(len(seq)) produce 0,1,...,len(seq)-1.
for i in range(len(seq)):
    print( seq[i] )
```

```
miao
32
casa
gatto
-7
1.2
```

Perché usare il secondo metodo? Sembra più complicato… Il secondo metodo permette più controllo sull'accesso alla sequenza. Conoscendo la sua posizione possiamo accedere, ad esempio, al suo successore, o al suo predecessore, ecc… Se poi la sequenza è una lista, possiamo anche **modificare** l'elemento `seq[i]`.

Esercizio: riscrivere i programmi di questa sezione ciclando sulle posizioni nella sequenza.

0.1.3 Riassumendo

Abbiamo visto: - La sintassi il ciclo `for`. - Esempi di uso : 1. trasformazione di una sequenza; 2. statistiche e conteggi su una sequenza; 3. filtraggio di dati in una sequenza; 4. generazione di sequenze. - Ciclo con accesso alla sequenza attraverso l'indice di posizione.