Istruzioni di Iterazione

Istruzioni di Iterazione

Le istruzioni di iterazione permettono di ripetere una o più istruzioni

Sono uno dei fondamenti della programmazione strutturata

- Insieme alle istruzioni condizionali
- ...Ed alla esecuzione in sequenza (naturale in Python)

Sono sufficienti a descrivere qualsiasi algoritmo

In Python ce ne sono due:

- I cicli `for` (di gran lunga i più usati)
  - Permettono di iterare sugli elementi di una collezione
- I cicli `while`
  - Permettono di iterare fintanto che una condizione è vera


Cicli `for` in Python

L'istruzione `for` (o ciclo `for`) in Python

...Permette di iterare sugli elementi di una collezione

- La sintassi è la seguente:

```
for <var> in <espr. collezione>:
    <blocco>
```

- Innanzitutto, viene definita la variable `<var>`
- Le istruzioni in `<blocco>` vengono ripetute per ogni elemento in `<collezione>`
- Ad ogni iterazione, a `<var>` viene assegnato come valore tale elemento


Cicli `for` in Python

Vediamo un semplice esempio:

In [9]:
```
for a in [1, 2, 3]:
    print(f'a: {a}')
```

```
a: 1
a: 2
a: 3
```

- Il blocco esegue una volta per ogni valore della lista
- Ad ogni iterazione la variabile `a` prende il valore di un elemento

La collezione può essere ottenuta con una qualunque espressione:

In [10]:
```
L = [1, 2, 3]
for a in L: # In questo caso abbiamo usato una variabile
    print(f'a: {a}')
```

```
a: 1
a: 2
a: 3
```


Cicli `for` in Python

Funziona con qualunque tipo di collezione, e.g. con le tuple:

In [11]:
```
for b in (4, 5, 6):
    print(f'b: {b}')
```

```
b: 4
b: 5
b: 6
```

...Ma anche con gli insiemi!

In [12]:
```
for c in {7, 8, 9}:
    print(f'c: {c}')
```

```
c: 8
c: 9
c: 7
```

- Gli insiemi sono però collezioni non ordinate
- Quindi non c'è controllo sull'ordine in cui gli elementi sono considerati

Cicli `for` in Python

Un ciclo su una collezione vuota non effettua alcuna iterazione:

In [13]:
```
for a in []:
    print('Il blocco esegue')
```

Iterare su un dizionario equivale ad iterare sulle chiavi

In [14]:
```
d = {'nome': 'Mario', 'cognome': 'Rossi'}
for k in d:
    print(k, ':', d[k])
```

```
nome : Mario
cognome : Rossi
```

- È poi possibile utilizzare la chiave
- ...Per accedere ad un valore nel dizionario


Cicli `for` e Variabili

Come nel caso delle istruzioni condizionali

...Ogni variabile definita nel blocco è come se fosse definita all'esterno:

In [15]:
```
for a in [1, 2, 3]:
    c = 2
print(f'c: {c}')
print(f'a: {a}')
```

```
c: 2
a: 3
```

- Ad ogni iterazione in `c` viene inserito il valore 2
- Alla fine `c` è disponibile all'esterno del ciclo
- Anche `a` conta come una variabile definita nel ciclo!
- Alla fine è disponibile all'esterno
- ...E contiene l'ultimo valore che le è stato assegnato nel ciclo

Cicli `while` in Python

L'istruzione `while` (o ciclo `while`) in Python

...Permette di iterare fintanto che una condizione è vera

- La sintassi è la seguente:

```
while <espressione>:
    <blocco>
```

- Ad ogni iterazione l'espressione viene valutata
- Se è vera, si esegue il blocco

Seppure più semplice, l'istruzione `while` è più flessibile del `for`

- Nella maggior parte dei casi, un `for` è sufficiente (e consigliato)
- ...Ma in alcuni casi di nicchia usare `while` è necessario

Cicli `while` in Python

Caso tipico: se non è noto a priori il numero di iterazioni da effettuare

- E.g. approssimare il valore della somma di una serie geometrica
- ...Fermandosi quando ogni modifica diventa più piccola di 10^{-20}

In [16]:
```
s, s_old = 0, -1
r, n = 0.4, 0
while abs(s - s_old) > 1e-20:
    s_old = s # memorizzo il vecchio valore di s
    s += r**n # aggiorno s
    n += 1    # incremento n
print(f'Valore finale di s: {s:.15f}')
print(f'Num. di iterazioni effettuate: {n}')
```

```
Valore finale di s: 1.666666666666667
Num. di iterazioni effettuate: 42
```

- Usiamo una variabile `s_old` per memorizzare il valore precedente della somma
- ...E la confrontiamo con `s` per capire quando fermarci


Cicli Infiniti

Occorre prestare attenzione a non creare cicli "infiniti"

...Ossia cicli in cui la condizione di proseguimento è sempre vera:

In [17]:
```
s, s_old = 0, -1
r, n = 0.4, 0
while abs(s - s_old) > 1e-20:
    s_old = s # memorizzo il vecchio valore di s
    s += r**n # aggiorno s
    #n += 1    # NOTA: "dimentico" di aggiornare n
print(f'Valore finale di s: {s:.15f}')
print(f'Num. di iterazioni effettuate: {n}')
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[17], line 3
      1 s, s_old = 0, -1
      2 r, n = 0.4, 0
----> 3 while abs(s - s_old) > 1e-20:
      4     s_old = s # memorizzo il vecchio valore di s
      5     s += r**n # aggiorno s

KeyboardInterrupt:
```

- Se vi capita su Jupyter, basta interrompere il kernel (pulsante "stop" in alto)

Programmazione Strutturata (e Non)

Come avevamo accennato:

Combinando istruzioni per sequenza, selezione ed iterazione è possibile esprimere qualsiasi algoritmo

Questo risultato è noto come teorema del programma strutturato

Detto questo a volte è comodo poter fare delle eccezioni

- Per questo, Python offre istruzioni che hanno effetto sul flusso di controllo
- ...Ma che non fanno parte della programmazione strutturata

Queste sono in particolare `break` e `continue`


Istruzione `break`

L'istruzione `break`, quando viene eseguita

...Interrompe l'istruzione di controllo di flusso corrente

In [18]:
```
l = [1, 5, 2, 6, 6, 8, 1]
da_trovare = 2
trovato = False
for v in l:
    if v == da_trovare:
        trovato = True
        break # interrompe il ciclo for
print(f'trovato: {trovato}')
```

```
trovato: True
```

Questo codice cerca un elemento in una lista

- Se l'elemento viene trovato, non è necessario continuare la ricerca
- ...E si può interrompere il ciclo con l'istruzione `break`

Istruzione `break`

L'istruzione `break`, quando viene eseguita

...Interrompe l'istruzione di controllo di flusso corrente

In [19]:
```
l = [1, 5, 2, 6, 6, 8, 1]
da_trovare = 2
trovato = False
for v in l:
    if v == da_trovare:
        trovato = True
print(f'trovato: {trovato}')
```

```
trovato: True
```

Se il break viene omesso (come sopra)

- Tutto funziona perfettamente
- ...Ma il codice è un po' più inefficiente

Istruzione `continue`

L'istruzione `continue`, quando viene eseguita in un ciclo

...Salta immediatamente all'iterazione successiva

In [20]:
```
l = [1, 5, 2, 6, 6, 8, 11]
massimo_pari = None
for v in l:
    if v % 2 == 1: # "salto" i numeri dispari
        continue
    if massimo_pari == None or massimo_pari < v:
        massimo_pari = v
print(f'Il massimo dei valori pari è: {massimo_pari}')
```

```
Il massimo dei valori pari è: 8
```

Questo codice trova il massimo dei numeri pari in una lista

- Se il numero corrente è dispari, viene eseguita una `continue`
- ...E l'esecuzione salta immediatamente all'iterazione successiva

Istruzione `continue`

L'istruzione `continue`, quando viene eseguita in un ciclo

...Salta immediatamente all'iterazione successiva

In [21]:
```
l = [1, 5, 2, 6, 6, 8, 1]
massimo_pari = None
for v in l:
    if v % 2 == 0: # considero solo i valori pari
        if massimo_pari == None or massimo_pari < v:
            massimo_pari = v
print(f'Il massimo dei valori pari è: {massimo_pari}')
```

```
Il massimo dei valori pari è: 8
```

Senza `continue` (come sopra)

- Si può ottenere lo stesso effetto usando un `if`
- ...Ma il codice diventa più innestato (e quindi un po' meno leggibile)