Corso di **Python**

Lezione 3

Istruzioni e funzioni built-in

Editor: Davide Brunato
Scuola Internazionale Superiore di Studi Avanzati di Trieste

Istruzioni di base

- Istruzioni per blocchi condizionali:
  - `if`
  - `elif`
  - `else`
- Istruzioni per cicli iterativi:
  - `while/while-else`
  - `for/for-else`
  - `break`
  - `continue`
- `import`
- `assert`
- `pass`
- `input (raw_input)`

L'istruzione **if-elif-else**

- È il controllo condizionale unificato di Python:
  - Oltre ad implementare il classico costrutto *if-then-else* è anche un equivalente del costrutto *switch* presente in altri linguaggi
  - Alla prima scelta **if** può infatti seguire una sequenza di ulteriori condizioni espresse con **elif** ed eventualmente anche un **else** finale
- Sintassi:

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

L'istruzione **while**

- L'istruzione **while** definisce un ciclo iterativo basato su una condizione
- Il ciclo continua fino a quando la condizione è True
- Sintassi:

```
while expression:
    statement(s)
[else:
    statement(s)]
```

- Il blocco **else** è opzionale e viene eseguito quando la condizione del ciclo diventa False
  - Se invece il ciclo viene interrotto esplicitamente (istruzione **break**) allora il blocco *else* non viene eseguito

L'istruzione **for**

- L'istruzione **for** definisce un ciclo iterativo ripetuto per tutti gli elementi di un'espressione *iterabile*
- Sintassi:

```
for target in iterable:
    statement(s)
[else:
    statement(s)]
```

- Il blocco **else** è opzionale e viene eseguito dopo la conclusione dell'iterazione
  - Se il ciclo viene interrotto esplicitamente da un'istruzione **break** allora il blocco *else* non viene eseguito
- Durante il for loop non è possibile alterare l'oggetto *iterabile* su cui si effettuano le iterazioni

Istruzioni per cicli iterativi

- Istruzione **`break`** : interrompe il ciclo iterativo
  - Esempio:

```
while True:
    if x > 1000: break
    x += 1
```

  - Se l'istruzione non è contenuta in un ciclo viene generato un errore:

```
>>> break
  File "<stdin>", line 1
SyntaxError: 'break' outside loop
```

- Istruzione **`continue`** : passa alla prossima iterazione del ciclo
  - Esempio:

```
for i in range(1,100):
    if i % 10: continue
    print(i)
```

  - Se l'istruzione non è contenuta in un ciclo viene generato un errore:

```
>>> continue
  File "<stdin>", line 1
SyntaxError: 'continue' not properly in loop
```

Uso del ciclo for

- Si può iterare con degli indici:

```
elenco = ['uno', 'due', 'tre']
for i in range(0, len(elenco)):
    print(elenco[i])
```

- Ma è più naturale iterare direttamente sulla lista:

```
elenco = ['uno', 'due', 'tre']
for elemento in elenco:
    print(elemento)
```

- Il range ha senso quando è necessario produrre la sequenza numerica:

```
for valore in range(1, 10):
    print(valore)
```

- Sui dizionari l'iterazione avviene sulla chiave:

```
tabella = {'primo': 20, 'secondo': 10, 'terzo': 5}
for chiave in tabella:
    print(tabella[chiave])
```

L'istruzione **import**

- Permette di importare moduli dove sono definite altre variabili, funzioni e classi:
  - L'import può avere in oggetto un qualsiasi file sorgente in Python
  - L'importazione può essere fatta nel path definiti per l'ambiente di esecuzione di Python o rispetto alla directory ove è collocato il file sorgente

- Esempio:

```
>>> import sys
>>> sys.subversion
('CPython', '', '')
>>> sys.version_info
sys.version_info(major=2, minor=7, micro=5, releaselevel='final',
serial=0)
>>> sys.version_info.major
2
>>> sys.exit
<built-in function exit>
```

L'istruzione **assert**

- L'istruzione `assert` permette di verificare una condizione e generare un'errore (eccezione) in caso di non soddisfacimento

  assert *expression* ["," *expression*]

- Istruzione molto usata nell'ambito dei test di applicazioni
- Esempio:

```
>>> a = 10
...
>>> assert a < 0, "'a' deve essere negativo!"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: 'a' deve essere negativo!
```

L'istruzione **pass**

- L'istruzione **pass** non fa nessuna azione
  - Serve a riempire i blocchi che non contengono istruzioni
  - I blocchi infatti devono contenere almeno un'istruzione
- Esempio:

```
if x < 0:
    x += 1
elif x >= 0 and x <= 10:
    pass
else:
    x -=1
```

- Può essere omessa nella dichiarazione di classi e funzioni quando viene specificata la *docstring*
- Mettere *pass* dove non serve non causa comunque errori né codice

Funzioni per l'input

- Per un input non interpretato si usa la funzione built-in **raw_input**

```
>>> s = raw_input()
Ciao
>>> s
'Ciao'
>>> s = raw_input("IN> ")
IN> Ciao Ciao
>>> s
'Ciao Ciao'
```

- In alternativa, se si deve inserire un'espressione da valutare, si può usare la funzione **input**
  - Si possono specificare variabili e funzioni da valutare
  - Equivale alla chiamata `eval(raw_input())`, dove eval() è una funzione built-in che valuta un'espressione espressa in formato stringa
- In Python 3.2 `raw_input` è stata ridenominata come input e la vecchia input è ora sostituibile con `eval(input())`

Programma di esempio con uso di ciclo while e if

Indovinare un numero tra 1 e 20

```
#!/usr/bin/env python
import random
guesses_made = 0

name = raw_input('Hello! What is your name?\n')
number = random.randint(1, 20)
print('Well, {0}, I am thinking of a number between 1 and 20.'.format(name))

while guesses_made < 6:
    guess = int(raw_input('Take a guess: '))
    guesses_made += 1
    if guess < number:
        print('Your guess is too low.')
    elif guess > number:
        print('Your guess is too high.')
    else:
        break

if guess == number:
    print('Good job, {0}! You guessed my number in {1} guesses!'.format(name, guesses_made))
else:
    print('Nope. The number I was thinking of was {0}'.format(number))
```

Run dell'esempio while + if

```
$ vi tmp/guessnum.py
$ chmod a+x tmp/guessnum.py
$ ./tmp/guessnum.py   # oppure "python tmp/guessnum.py"
Hello! What is your name?
Davide
Well, Davide, I am thinking of a number between 1 and 20.
Take a guess: 10
Your guess is too high.
Take a guess: 7
Your guess is too high.
Take a guess: 4
Your guess is too low.
Take a guess: 5
Good job, Davide! You guessed my number in 4 guesses!
```

Dettagli per eseguire script

- Come per gli altri linguaggi di scripting in Unix è attivo il riconoscimento della sequenza iniziale `#!` (cosiddetta *shebang*) per definire l'interprete dello script
- In Python si usa la specifica diretta:

  ```
  #!/usr/bin/python
  ```

  o quella indiretta, che richiama il primo interprete che viene trovato nel `PATH` di sistema:

  ```
  #!/usr/bin/env python
  ```

- In generale è bene specificare l'interprete solo per gli script che si vogliono eseguire direttamente via CLI, mentre sugli altri file sorgenti è meglio non includere nulla, per evitare confusione sull'interprete da utilizzare
- La forma esplicita ha la precedenza sulla *shebang*:

```
$ python3 tmp/guessnum.py
Traceback (most recent call last):
  File "tmp/guessnum.py", line 5, in <module>
    name = raw_input('Hello! What is your name?\n')
NameError: name 'raw_input' is not defined
```

Funzioni built-in per sequenze

- Alcune funzioni base applicabili sulle sequenze:
  - len
  - range
  - reversed
  - sorted
- Altre funzioni applicabili alle sequenze le vedremo più avanti (*slice*, iteratori, ...)

La funzione **len**

- Ritorna la lunghezza (numero di elementi) di un oggetto:

```
len(s)
```

- L'argomento può essere una sequenza (stringa, bytes, tupla, lista, o prodotto di funzioni come range) o una collezione (dizionario, insieme o frozenset)
  - In generale è applicabile a tutti gli oggetti, ad esclusione dei tipi di dati semplici
  - Vedremo che per le classi è definita mediante metodo `__len__`
- Esempi:

```
>>> a = [10, 20, 30]
>>> len(a)
3
>>> s = "Questa è una stringa"
>>> len(s)
21
>>> len(range(10))
10
```

La funzione **range**

- **range** è una funzione built-in utile per costruire una sequenza di numeri interi
- E' usata spesso nei cicli for quando si vuole ripetere il ciclo un preciso numero di volte sulla base di valori numeri predefiniti
- Esempi (interprete python 2.7):

```
>>> range(10)                   # Parte da 0 per default
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0, 10)                # Specifico un intervallo
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10, 2)             # Specifico intervallo e step
[1, 3, 5, 7, 9]
>>> range(1, 10, 3)
[1, 4, 7]
>>> range(5, 0, -1)             # Anche all'indietro …
[5, 4, 3, 2, 1]
```

- In Python 3 la funzione `range` è stata unificata con la funzione built-in `xrange`, che al posto di una lista generava un iteratore. In pratica xrange è stata ridenonimata come range e per ottenere la lista equivalente si applica il costruttore:

```
>>> range(10)
range(0, 10)
>>> type(range(10))
<class 'range'>
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Funzione **reversed**

- Questa funzione inverte una sequenza
  - Non genera una sequenza nuova ma un *iteratore* che può essere usato in un ciclo iterativo
  - Dalla versione 2.6 è associabile a generici oggetti che implementino il metodo `__reversed__`
- Esempio:

```
>>> lista = [1, 2, 3, 4, 5]
>>> reversed(lista)
<list_reverseiterator object at 0x7fe4d554e780>
>>> for i in reversed(lista):
...     print(i)
...
5
4
3
2
1
```

Funzione **sorted**

- Questa funzione produce una lista ordinata a partire da un oggetto *iterabile*

```
sorted(iterable[, cmp[, key[, reverse]]])
```

- Esempio:

```
>>> lista = [20, 10, -2, 40, 0]
>>> sorted(lista)
[-2, 0, 10, 20, 40]
>>> lista
[20, 10, -2, 40, 0]
```

- Per una lista c'è anche la funzione sort(), ma è diversa:

```
>>> lista = [-10, 20, "Uno", None]
>>> lista.sort()
>>> lista
[None, -10, 20, 'Uno']
```

Funzione **sorted**

- Questa funzione produce una lista ordinata a partire da un oggetto *iterabile*

```
sorted(iterable[, cmp[, key[, reverse]]])
```

- Esempio:

```
>>> lista = [20, 10, -2, 40, 0]
>>> sorted(lista)
[-2, 0, 10, 20, 40]
>>> lista
[20, 10, -2, 40, 0]
```

- Per una lista c'è anche la funzione sort(), ma è diversa:

```
>>> lista = [-10, 20, "Uno", None]
>>> lista.sort()
>>> lista
[None, -10, 20, 'Uno']
```

Funzioni built-in per attributi

- Ogni oggetto Python ha degli *attributi*, ai quali si accede mediante notazione *punto*:

```
>>> x = -10
>>> dir(x)
['__abs__', …, 'conjugate', 'denominator', 'imag', 'numerator', 'real']
>>> x.__abs__()    # Equivalente di abs(x)
10
```

- In alternativa per la gestione degli attributi ci sono delle funzioni built-in:
  - **getattr**(*object, name[, default]*): per ottenere l'attributo

```
>>> getattr(x, 'numerator')
  10
```

    - Accedere ad attributi non esistenti genera un errore `AttributeError:`

```
>>> getattr(x, 'numeratore')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'numeratore'
```

  - **hasattr**(*object, attribute_name*): per testare se un oggetto ha un certo attributo
  - **setattr**(*object, attribute_name, value*): per modificare o aggiungere attributi

Cancellazione di oggetti

- Per rimuovere oggetti in Python si usa l'istruzione **del**:

```
>>> a = 10
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

- Con **del** di possono rimuovere singoli oggetti di liste o dizionari:

```
del lista[10]
del mappa['chiave']
```

- Per la rimozione di attributi si può usare sempre l'istruzione **del** oppure la funzione built-in **delattr**:

```
delattr(object, name)
```