Corso di **Python**

Lezione 4

Le espressioni

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

Espressioni

- Un'espressione è una parte di codice che produce un risultato o un valore
- Elementi che possono comprorre espressioni in Python:
  - Accesso agli attributi di oggetti e funzioni
  - Accesso agli elementi di una sequenza
  - *Slicing*
  - Chiamata di funzione
  - Espressioni *lambda*
  - Espressioni condizionali
  - Altri operatori caratteristici
  - *List comprehension*
  - Espressioni generatrici

Atomi

- Sono gli elementi base che compongono un'espressione, collegati tra loro dagli operatori
- Un **atomo** di espressione può essere un:
  - Identificatore (variabile, *funzione*)
  - Letterale (stringhe, numeri, False, True, None)
  - Aggregato (enclosure)
- L'**aggregato** è un costrutto delimitato da parentesi, che assume diversi significati in base alle parentesi usate e alla forma interna:
  - Parentesi **tonde** per racchiudere altre espressioni (uso classico, identico a quello fatto in C++/Java), una tupla di espressioni, o un'espressione *generatrice*
  - Parentesi **quadre** per esprimere liste
  - Patentesi **graffe** per esprimere dizionari o insiemi (da Python 2.7+)
- Gli aggregati sono gli operatori a maggiore priorità

Accesso agli elementi di una sequenza

- Per accedere agli elementi di sequenze si usa la tipica notazione basata su parentesi quadre (subscription)
- Per stringhe, liste e tuple si usa un indice numerico:

```
>>> a = [10, 20, 30]
>>> a[0]
10
>>> a[-1]
30
```

- Per i dizionari si usa la chiave:

```
>>> d = {'alpha': 1, 'beta': 2}
>>> d['alpha']
1
```

Slicing

- Letteralmente *affettare* un oggetto a sequenza ordinata (stringa, tupla o lista)
- Vi sono due sintassi previste per le slice:

  obj[*start*:*stop*]

  obj[*start*:*stop*:*stride*] # stride significa *passo*

  start, *stop* e *stride* sono espressioni opzionali

- Esempi:

```
>>> a = [10, 20, 30, 40, 50]
>>> a[:]
[10, 20, 30, 40, 50]
>>> a[0:]
[10, 20, 30, 40, 50]
>>> a[-11:20]              # Start e stop possono sforare i limiti della sequenza
[10, 20, 30, 40, 50]
>>> a[:-1]
[10, 20, 30, 40]
>>> a[::2]
[10, 30, 50]
>>> a[::-1]                # Inversione
[50, 40, 30, 20, 10]
```

Funzione built-in **slice**

- Permette di definire uno oggetto *slice* per affettare sequenze:

  slice(*stop*)

  slice(*start*, *stop*[, *step*])

  - Il vantaggio rispetto al costrutto esplicito è che lo si può definire una volta sola e utilizzarlo più volte

- Esempio:

```
>>> a = range(10)
>>> a[:5]
[0, 1, 2, 3, 4]
>>> a[slice(5)]
[0, 1, 2, 3, 4]
>>> a[slice(3, 6)]   # Equivalente a a[3:6]
[3, 4, 5]
>>> a[slice(3, None)]  # Equivale a a[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> a[slice(3, None, 2)]  # Equivale a a[3::2]
[3, 5, 7, 9]
```

Espressioni lambda

- Permettono di costruire piccole funzioni da una singola espressione, utili per costruire un codice più contenuto
- Sintassi:

```
lambda parametri: espressione
```

- Esempio:

```
>>> import math
>>> square_root = lambda x: math.sqrt(x)
>>> square_root(4)
2.0
>>> sum = lambda x, y: x + y
>>> sum(4, 7)
11
>>> 1 + (lambda x, y: x + y)(4, 7)   # Esempio di uso immediato ...
12
```

Espressioni condizionali

- Da Python 2.5 è stato introdotto un operatore ternario per le espressioni condizionali:

  ```
  expr1 if condition else expr2
  ```

  - Simile al costrutto *"e1 ? cond : e2"* presente in C++ e Java
  - Si possono annidare anche senza usare di parentesi perché è un operatore a minore priorità

- Esempi:

```
>>> a = 10 if b > 0 else 20
>>> a = 10 if b > 0 else 20 if b > 10 else 30

>>> name = 'Pippo'
>>> print('Hello, ' + (name if name else 'Anonymous'))
Hello, Pippo
>>> name = ''
>>> print('Hello, ' + (name if name else 'Anonymous'))   # Parentesi!
Hello, Anonymous
```

Espressioni condizionali booleane

- In Python si possono costruire *espressioni condizionali booleane* essendo che ogni dato può essere trattato come valore di verità e che gli operatori ***or*** e ***and*** ritornano l'ultimo argomento valutato
- **`x or y`** `# Ritorna y se x è False, x altrimenti`

```
>>> print([] or None)
None
>>> print([1, 2] or None)
[1, 2]
>>> name = 'Pippo'
>>> print('Hello, ' + (name or 'Anonymous'))
Hello, Pippo
>>> name = ''
>>> print('Hello, ' + (name or 'Anonymous'))
Hello, Anonymous
```

- **`x and y`** `# Ritorna x se x è False, y altrimenti`

```
>>> completato = True
>>> print((completato and "Completato con successo!!") or "Non completato!!!")
Completato con successo!!
>>> completato = False
>>> print((completato and "Completato con successo!!") or "Non completato!!!")
Non completato!!!
```

L'operatore **in - not in**

- È l'operatore per il test di appartenenza che si applica a tutti gli oggetti contenitori:

```
item [not] in container_object
```

- Per liste, tuple e insiemi verifica la presenza dell'elemento:

```
>>> t, s = ("Ciao", 1, False), {None, "Alpha", "Beta"}
>>> 1 in t, None in s
(True, True)
>>> 2 in t, "Gamma" in s
(False, False)
```

- Per i dizionari il test di appartenenza è verificato se l'elemento è tra le chiavi del dizionario:

```
>>> d = {"Uno": 1, "Due": 2, "Tre": 3}
>>> "Uno" in d
True
>>> "Quattro" in d
False
```

- Per le stringhe il test è verificato se l'elemento è una sottostringa:

```
>>> "Linux" in "CentOS Linux release 7.1.1503 (Core)"
True
```

L'operatore **is** - **is not**

- Si usa per testare l'identità degli oggetti
- **`a is b`** : è True se e solo se a e b sono lo stesso oggetto:

```
>>> a = 1
>>> a is None
False
>>> b = None
>>> b is None
True
```

- **`a is not b`** : è la negazione di **`a is b`** (meglio di: **`not a is b`**)
- La comparazione *is* - *is not* è comunque preferibile quando dobbiamo comparare un oggetto con un *singleton non numerico* (None, False o True) altrimenti sono da preferire gli operatori `==` o `!=`

Precedenza degli operatori

Operatore	Descriptione
`(expr...), [expr...], {key: value...}, {expr...}`	Aggregati (*operatori a maggiore priorità*)
`x[i], x[i:i], x(args...), x.attribute`	Sottoscrizione, slicing, call, accesso ad attributo
`**`	Elevazione a potenza
`+x, -x, ~x`	Segno, NOT a bit
`*, @, /, //, %`	Moltiplicazione, moltiplicazione a matrice, divisione, modulo
`+, -`	Addizione e sottrazione
`<<, >>`	Shift a livello di bit
`&`	AND a bit
`^`	XOR a bit
`|`	OR a bit
`in, not in, is, is not, <, <=, >, >=, !=, ==`	Comparazioni, appartenenza e identità
`not x`	NOT booleano
`and`	AND booleano
`or`	OR booleano
`if - else`	Espressione condizionale
`lambda`	Espressioni lambda

List comprehension

- Dalla versione 2.0 Python prevede un costrutto per creare una lista da un'espressione iterabile denominato comunemente **list comprehension**:

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Un codice equivalente sarebbe:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Si può mettere anche una condizione, per considerare solo alcuni elementi:

```
>>> [x**2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

- Al posto del costrutto `[ ]` si può equivalentemente usare la funzione built-in *list*:

```
>>> list(x**2 for x in range(10) if x % 2 == 0)
[0, 4, 16, 36, 64]
```

List comprehension annidate

- Per generare delle matrici si possono annidare più list comprehension:

```
>>> [[ 1 if row == col else 0 for row in range(3)] for col
in range(3)]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

- Si può anche usare la list comprehension per trasporre gli elementi di una matrice esistente:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Altri tipi di *comprehension*

- Da Python 2.7+ è definita la possibilità di definire dei dizionari o degli insiemi in maniera del tutto simile alla list comprehension:

```
>>> {n: n**2 for n in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
>>> {n**2 for n in range(5)}
set([0, 1, 4, 16, 9])
```

- Su versioni precedenti si poteva ovviare con funzioni built-in:

```
>>> dict((n, n**2) for n in range(5))
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
>>> set(n**2 for n in range(5))
set([0, 1, 4, 16, 9])
```

Espressioni generatrici

- Le espressioni generatrici (generator expressions) sono la forma concisa per definire un *generatore*
  - Un generatore è una forma particolare di iteratore, che si avvale di un tipo particolare di controllo (vedremo l'istruzione *yield* quando si parlerà di funzioni)
  - L'espressione generatrice viene poi usata naturalmente dal ciclo **for** o con l'uso della funzione **next**
- Per definire un'espressione generatrice ci si avvale di una sintassi con parentesi tonde:

```
g = (x ** 2 for x in range(10))
```

- Le variabili definite in un'espressione generatrice sono valutate in modo *lazy* (non immediato), tranne quelle relative al ciclo for più esterno (quello più a sinistra)

  Esempio:

```
>>> g1 = (x*y for x in range(10) for y in bar(x))
>>> next(g1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <genexpr>
NameError: name 'bar' is not defined
```

Novità di Python 3.5

- In Python 3.5 è stato introdotto un operatore di moltiplicazione tra matrici che usa il simbolo '**@**'
  - Per ora questo operatore non funziona sui tipi base ma è possibile implementarlo sulle classi definendo appositi metodi (*__matmul__()*, *__rmatmul__()* e *__imatmul__()*)
  - Verrà implementato in modo completo in versioni successive
  - L'operatore è stato introdotto per semplificare la scrittura di questo tipo di espressioni:

```
S = (H @ beta - r).T @ inv(H @ V @ H.T) @ (H @ beta - r)
```

```
S = dot((dot(H, beta) - r).T, dot(inv(dot(dot(H, V), H.T)),
dot(H, beta) - r))
```

  - Il nuovo operatore è già utilizzabile con la libreria *numpy* 1.10

Funzioni built-in utilizzabili nelle espressioni

- all
- any
- min
- max
- sum
- pow
- abs
- round
- eval

Funzioni **all** e **any**

- Applicano il test ad un'intera sequenza (un *iterabile*)
- Introdotte in Python 2.5
- **`all`**`(`*`iterable`*`)`

  Ritorna True se tutti gli elementi della sequenza iterabile sono True o se l'iterabile è vuoto. Equivalente alla seguente funzione:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

- **`any`**`(`*`iterable`*`)`

  Ritorna True se almeno uno degli elementi della sequenza iterabile è True. Se l'iterabile è vuoto ritorna False. Equivalente alla seguente funzione:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

Funzioni **min** e **max**

- Calcolano il minimo e il massimo di una sequenza iterabile o elenco di argomenti
- `min(`*`iterable`*`[, `*`key`*`])`

  `min(`*`arg1`*`, `*`arg2`*`, `*`*args`*`[, `*`key`*`])`
  - L'iterabile non deve essere vuoto
  - L'argomento opzionale deve essere specificato come *keyword argument* è rappresenta una funzione di ordinamento per il confronto tra i valori
- Esempi:

```
>>> min(2, 3, -5, 6)
-5
>>> min((2, 3, -5, 6))
-5
>>> min(2, 3, (-5, 6))
2
>>> min(2, 3, *(-5, 6))
-5
>>> min(2, 3, -5, 6, key=lambda x:-x)
6
```

Funzione **sum**

- Somma degli elementi di un iterabile

  sum(*iterable*[, *start*])

  - Somma al valore start gli elementi dell'iterabile, da sinistra a destra e ritorna il totale
  - Il default per start è `0`
  - E' possibile sommare elementi numerici e sequenze, ma non stringhe …

- Esempi:

```
>>> sum((1, 2, 3))
6
>>> sum((1, 2, 3), -1)
5
>>> sum([(1, 1), (2, 1)], (0, 0))
(0, 0, 1, 1, 2, 1)
>>> sum(['Ciao ', 'ciao'], '')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() can't sum strings [use ''.join(seq) instead]
```

Funzione **pow**

- Ritorna la potenza di un numero:

  pow(*x*, *y*[, *z*])

  - Il valore ritornato è ***x*** elevato alla potenza ***y*** (la forma con due argomenti è equivalente a `x**y`)
  - Se *z* è presente allora ritorna ***x*** elevato alla potenza ***y*** modulo ***z*** (più efficiente di "`pow(x, y) % z`")
  - Gli argomenti devono essere numerici e il tipo ritornato dipende dagli argomenti
- Vincoli attuali su ***z***: deve essere intero e opera solo con x e y interi, y non negativo
- Esempi:

```
>>> pow(2, 2)
4
>>> pow(2, 2, 3)
1
>>> pow(2, 2, 3.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pow() 3rd argument not allowed unless all arguments are integers
>>> pow(2, 2) % 3.1
0.8999999999999999
```

Funzioni **abs** e **round**

abs(*x*)

- Ritorna il valore assoluto del numero;
- In caso di numero complesso ritorna il *modulo*

round(*number*[, *ndigits*])

- Ritorna il valore del numero arrotondato alla *ndigits*-*esima* cifra decimale
- Se *ndigits* è omesso ritorna numero intero più vicino
- Richiama la funzione interna all'oggetto:

```
number.__round__(ndigits)
```

Funzione **eval**

- Permette di valutare un'espressione espressa in forma di stringa come fosse effettivamente parte del codice

  eval(*expression*[, *globals*[, *locals*]])

  - L'argomento deve essere una stringa Unicode o Latin-1
  - *globals* e *locals* possono essere dizionari di valori utilizzati per valutare l'espressione

- Esempi:

```
>>> x = 1
>>> eval('x + 1')
2
>>> eval('x + 1', {'x': 10})
11
>>> eval('x + 1', {'x': 10}, {'x': -10})
-9
```