Corso di **Python**

Lezione 6

Le funzioni

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

Funzioni

- Si possono raggruppare le istruzioni in blocchi che posso essere richiamati su richiesta (*function call*)
  - Python ha una serie di funzioni predefinite (built-in)
  - Altre funzioni sono definite nelle librerie
  - L'utente ha la possibilità di definire le proprie funzioni con la sintassi:

```
def function-name(parameters):
    statement(s)
```

- Non è obbligatorio specificare dei parametri o far ritornare un valore (la funzione per default ritorna comunque *None*):

```
def nop():
    pass
```

Struttura di una funzione

```
def somma(a, b):
    return a + b
```

- **somma** è l'*identificatore della funzione*
- **a** e **b** sono i *parametri formali* della funzione:
  - I parametri sono separati da virgole
  - Non hanno una specifica di tipo (*dynamic typing*)
  - Nel chiamare la funzione è necessario specificare esattamente il numero di argomenti dichiarati:

```
>>> somma(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: somma() takes exactly 2 arguments (1 given)
```

- Valore/i di ritorno:
  - L'istruzione **return** permette di ritornare uno o più valori come risultato della funzione
  - I valori possono essere di un tipo qualsiasi, mutabile o immutabile, raggruppati eventualmente in un contenitore (es. lista) o in una tupla qualora li si separi semplicemente con delle virgole
  - Senza un return esplicito di un valore la funzione ritorna **None**

Tipizzazione dinamica

- La funzione **somma** definita in precedenza può essere chiamata con argomenti di tipo diverso:

```
>>> somma(2, 5)
7
>>> somma("super", "man")
'superman'
>>> somma(["a", 1, False], [5, 29, None])
['a', 1, False, 5, 29, None]
```

- Errori vengono generati se non è definito l'operatore di somma o se gli operandi non sono compatibili:

```
>>> somma({"a", 1, False}, {5, 29, None})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in somma
TypeError: unsupported operand type(s) for +: 'set' and 'set'
>>> somma(1, [5, 29, None])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sum
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Funzioni come oggetti

- Le funzioni possono essere assegnate come fossero oggetti mutabili:

```
>>> somma(1, 4)
5
>>> f = somma
>>> f(3, 9)
12
```

- Le funzioni possono essere passate come parametri ad altre funzioni:

```
>>> def cubo(x):
...     return x**3
...
>>> def chiama(f, arg):
...     return f(arg)
...
>>> chiama(cubo, 3)
27
```

Funzione built-in **callable**

- Per testare se un oggetto è una funzione si usa la funzione **callable**:
  - Questa funzione ritorna True se l'oggetto può essere chiamato come funzione, ritorna False in caso contrario
  - In generale sia le funzioni che le classi sono *callable* (mentre le istanze di una classe lo sono solo quando si definisce uno specifico metodo `__call__`)

```
>>> def f(x):
...     return x + 1
...
>>> callable(f)
True
>>> callable('alpha')
False
```

Questa funzione era stata tolta nella prima release di Python 3, ma è stata reinserita in modo definitivo dalla versione 3.2.

Parametri opzionali 1/2

- Dopo i *parametri formali* possono essere specificati dei ***parametri opzionali***:

```
def f(x, y=[]):
    y.append(x)
    return y
```

- Provando sull'interprete interattivo questa funzione notiamo un comportamento particolare:

```
>>> f(23)
[23]
>>> f(30)
[23, 30]
>>> f(50, [])
[50]
>>> f(23)
[23, 30, 23]
```

C'è un ***effetto memoria*** sul parametro opzionale perché il parametro è istanziato con un valore iniziale al momento della definizione della funzione

Parametri opzionali 2/2

- Per evitare l'*effetto memoria* è necessario definire il valore di default del parametro opzionale con un oggetto non mutabile:

```
def f(x, y=None):
    if y is None: y = []
    y.append(x)
    return y
```

- In questo caso il parametro opzionale viene comunque riassegnato:

```
>>> f(23)
[23]
>>> f(30)
[30]
>>> f(50, [40])
[40, 50]
>>> f(23)
[23]
```

Come valori di default è quindi bene usare sempre dei valori immutabili (spesso si trova None), usando valori mutabili solo quando il problema specifico lo richiede.

Argomenti con nome

- Il passaggio di argomenti può essere fatto associando nome del parametro con il valore, si parla in questo caso di *named arguments* o *keyword arguments*:

```
>>> def connect(a, b='', c=''):
...     return a + ' ' + b + ' ' + c
...
>>> connect("Uno", c="Tre", b="Due")
'Uno Due Tre'
>>> connect("Uno", c="Tre")       # Si specifica solo l'ultimo opzionale
'Uno  Tre'
>>> connect(c="Tre", a="Uno", b="Due")
'Uno Due Tre'
>>> connect(c="Tre", "Uno", b="Due")  # Argomenti formali solo all'inizio!
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

- Il passaggio di argomenti per nome è particolarmente utile per parametri opzionali, perché si può effettuare una chiamata di funzione valorizzando solo parte dei parametri opzionali

Argomenti variabili 1/2

- È possibile passare un numero variabile di parametri specificando un argomento preceduto da un'asterisco
  - All'interno della funzione tale argomento sarà invariabilmente una tupla
- Esempio:

```
def somma(*numeri):
    return sum(numeri)

>>> somma()
0
>>> somma(1)
1
>>> somma(1, 5, 4)
10
>>> args = (1, 5, 4)     # oppure args = 1, 5, 4
>>> somma(*args)
10
```

Argomenti variabili 2/2

- Si può passare un elenco variabile di argomenti con nome con il costrutto *`**identificatore`*, che raggrupperà gli argomenti passati in un dizionario:

```
>>> def f(**kwargs):
...     print(kwargs)
...
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
>>> d = { "a":1, "b":2 }
>>> f(**d)
{'a': 1, 'b': 2}
```

- Spesso si possono trovare i due costrutti accoppiati nella forma:

```
def f(*args, **kwargs):
    print("ARGS: ", args)
    print("KWARGS: ", kwargs)

>>> f(0, 5, a=1, b=2)
('ARGS: ', (0, 5))
('KWARGS: ', {'a': 1, 'b': 2})
```

Docstrings

- Ogni funzione può avere una stringa di documentazione:

```
def somma(*numeri):
    '''Somma i numeri passati come argomenti.'''
    return sum(numeri)
```

- Le *docstrings* sono definibili anche per moduli e classi
- Le stringhe di documentazione, a differenza dei commenti semplici, sono accessibili anche in runtime:

```
>>> somma.__doc__
'Somma i numeri passati come argomenti.'
```

- Il modulo *doctest* permette di utilizzare le stringhe di documentazione in modo unitario e coordinato
- Le docstrings hanno anche loro uno loro stile consigliato:
  - PEP 257 - https://www.python.org/dev/peps/pep-0257/

Function annotations

- Per ovviare alla tipizzazione dinamica in Python 3.5 sono state introdotte delle annotazioni di tipo (*type hints*) sulle funzioni:
  - https://www.python.org/dev/peps/pep-3107
  - https://www.python.org/dev/peps/pep-0484
- Definita nel modulo **typing**:
  - https://docs.python.org/3/library/typing.html
- Per le annotazioni intenzionalmente non è stata definita una semantica:
  - Nessun controllo di tipo automatico viene effettuato a runtime
  - Viene creato un attributo `__annotations__`
  - Usata solo per analisi off-line del codice
- Esempio:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Attributi delle funzioni 1/2

- Attributi standard delle funzioni:

```
>>> somma.__name__              # Nome della funzione
'somma'
>>> somma.__module__            # Modulo dove la funzione è definita
'__main__'
>>> f.__defaults__              # Defaults dei parametri opzionali
(None,)
>>> somma.__globals__           # Dizionario r/o con le variabili globali
{'somma': <function somma at 0x7f395178f320>, 'd': {'a': 1, 'b': 2}, 'f': <function f
at 0x7f395178f2a8>, '__builtins__': <module '__builtin__' (built-in)>, 'args': (1, 5,
4), '__package__': None, 'connect': <function connect at 0x7f395d471938>, '__name__':
'__main__', '__doc__': None}
>>> somma.__code__              # Codice oggetto della funzione
<code object somma at 0x7f395d4161b0, file "<stdin>", line 1>
>>> somma.__closure__           # Tupla con info degli scope interni (funzioni annidate)
```

- Attributi arbitrari:

```
>>> somma.__dict__              # Dizionario degli attributi arbitrari
{}
>>> somma.attr1 = 'attributo'
>>> somma.__dict__
{'attr1': 'attributo'}
```

Attributi delle funzioni 2/2

- Gli attributi, a parte `__globals__` e `__closure__`, sono tutti R/W
- Nota storica: in Python 2 diversi attributi standard avevano nome *`func_*`*
  - Esempio: `func_doc` invece di `__doc__`
  - Da Python 2.6 sono stati introdotti i nuovi nomi
  - I vecchi nomi degli attributi sono stati rimossi in Python 3
- In Python 3 sono stati introdotti altri attributi:

Attributo	Descrizione	Accesso
`__qualname__`	Nome qualificato (completo di modulo e classe)	R/W
`__annotations__`	Annotazioni dei parametri	R/W
`__kwdefaults__`	Defaults dei parametri keyword	R/W

I namespace

- Una funzione ha il suo spazio dei nomi locale (*local namespace*) detto anche *local scope*
- Le variabili definite all'interno di una funzione sono *variabili locali*
- Variabili definite a livello di file sorgente (modulo) sono invece dette *variabili globali*
  - ***Non esistono variabili globali in assoluto, sono sempre relative ad un modulo***
- Per riconnettersi a variabili globali all'interno di una funzione si usa la direttiva **global**:

```
_counter = 0
def count(incr=1):
    global _counter
    _counter += incr
    return _counter

>>> count()
1
>>> count(5)
6
```

- In Python 3 è stata definita la nuova direttiva **nonlocal**, che permette di rifeferirsi alla prima variabile con lo stesso nome in contesto esterno ma non globale
- L'uso di variabili globali, tipico di un modello procedurale, è in genere da evitare in favore di una metodologia object-oriented

Funzioni per i namespace

- Ci sono funzioni built-in che permettono di ispezionare i namespace:
  - **`globals`**`()` Ritorna un dizionario che rappresenta l'attuale tabella di simboli globali, relativamente al modulo dove viene effettivamente chiamata;
  - **`locals`**`()` Ritorna un dizionario che rappresenta l'attuale tabella di simboli locali. Se chiamata all'interno di una funzione ritorna le variabili definite localmente, compresi gli argomenti passati alla funzione;
  - **`vars`**`([`*`object`*`])` Ritorna il l'attributo `__dict__` dell'oggetto passato (eccezione di errore se l'oggetto non ha un attributo `__dict__`). Funziona come locals se viene passata senza argomento.

- I dizionari ritornati possono essere anche modificati, ma le modifiche funzionano solo per le variabili globali, ossia quelle a livello di modulo (se si modifica il dizionario *locals()* di una funzione le modifiche non sono viste dall'interprete …):

```
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> globals()['b'] = 10     # Equivalente a 'b = 10', ma non si usa ...
>>> b
10
```

Funzioni generatrici

- In Python è possibile definire delle funzioni generatrici (*generators*)
- Quando si chiama una funzione generatrice il corpo della funzione non viene eseguito
- Viene invece ritornato uno speciale iteratore che ingloba il corpo della funzione, le variabili locali e il punto di esecuzione, che inizialmente è l'inizio della funzione
- Per specificare un generatore si usa l'istruzione **yield**:

```
yield expression
```

- Il generatore viene poi usato richiamandolo con la funzione built-in **next**:
  - Con yield l'esecuzione della funzione viene congelata (puntatore di esecuzione corrente, variabili locali) e viene ritornato il valore dell'espressione al metodo *next* chiamante
  - Quando si chiama il successivo metodo *next* il controllo sul generatore l'esecuzione riprende da dove è stata interrotta, fino alla prossima istruzione *yield*
  - Se si termina la funzione normalmente allora l'iterazione si conclude mediante eccezione **StopIteration**
- Esempio di generatore:

```
def fibonacci(n, start=1):
    a, b = start, start + 1
    while a < n:
        yield a
        a, b = b, a + b
```

Ricorsione

- Python supporta la ricorsione ma c'è un limite sulla profondità delle chiamate ricorsive:

```
>>> import sys
>>> sys.getrecursionlimit()
1000
```

- Se si eccede questo limite viene generata un'eccezione (*RecursionLimitExceeded*):

```
>>> def fib(n, sum):
...     return sum if n < 1 else fib(n-1, sum + n)
...
>>> fib(998, 0)
Traceback (most recent call last):
  File "<stdin>", line 5, in fib
  ...
  File "<stdin>", line 5, in fib
RuntimeError: maximum recursion depth exceeded in comparison
```

- Per modificare questo limite:

```
>>> sys.setrecursionlimit(2000)
```

Funzioni annidate

- È possibile definire funzioni annidate in altre funzioni
- La funzione annidata può accedere alle variabili definite in quella esterna:

```
def percentuali(a, b, c):
    def vp(x, total=a+b+c)
         return (x*100.0) / total
    print('Percentuali: ', vp(a), bp(b), vp(c))
```

- La funzione annidata può essere progettata per costruire una funzione parametrica da ritornare come risultato della funzione:

```
def make_adder(augend):
    def add_k(addend):
        k = augend
        return addend + k
    return add_k
```

- Una funzione annidata utilizza variabili locali della funzione esterna viene detta ***closure***
  - La closure viene salvata nell'oggetto funzione ritornato (attributo `__closure__`), con una tupla dei valori utilizzati (memorizzati in cosiddette *celle*)
  - Le closure sono una violazione dei meccanismi tipici della programmazione object-oriented, ma alle volte servono

Decoratori

- I decoratori sono funzioni che prendono in argomento una funzione e la modificano
- Per specificare un decoratore si usa il costrutto:

```
@some_decorator
def some_function():
    # function body…
```

che equivale al seguente:

```
def some_function():
    # function body…
some_function = some_decorator(some_function)
```

- Definizione della sintassi:
  - https://www.python.org/dev/peps/pep-0318/
- Un'introduzione pratica ai decoratori:
  - http://programmingbits.pythonblogs.com/27_programmingbits/archive/50_function_decorators.html

I decoratori sono molto usati per modificare *classi*, metodi e *attributi*, per i quali Python prevede una serie di decoratori predefiniti.

Riassunto sulle funzioni 1/2

- In Python le funzioni ritornano sempre un valore, None nei casi in cui la funzione termina senza un return esplicito
- Non serve mettere assegnare il valore di ritorno ad una variabile, senza assegnamento il risultato viene semplicemente scartato
- Si può ritornare un qualsiasi oggetto, anche un oggetto mutabile creato all'interno della funzione come oggetto locale
- Il valore di ritorno è dinamico, non serve quindi definire il tipo di dato ritornato dalla funzione (int, float, void, ...)
- In Python il passaggio di argomenti variabili si realizza con due costrutti simili ed assimilabili a strutture dati base del linguaggio:

  `*` : passaggio di argomenti (es. `f(*args)`) assimilabile ad una **tupla**

  `**` : passaggio di argomenti con nome (es: `f(**kwargs)`) assimiliabile a un **dizionario**
- La possibilità di specificare *named arguments* è utile per passare solo alcuni parametri opzionali

Riassunto sulle funzioni 2/2

- In Python non c'è la distinzione tra passaggio per *valore* o *riferimento*, che costringe a due sintassi differenti, come ad esempio in PHP:

```
function swap($a, $b)
function swap(&$a, &$b)
```

- Il passaggio di parametri avviene sempre e solo per assegnamento:

```
def swap(s1, s2):
    return s2, s1

a, b = swap(a, b)        # Anche se in questo caso basterebbe 'a, b = b, a'
```

- La possibilità di modificare i parametri passati sussiste solo per parametri *mutabili* (ad esempio una lista o un dizionario):

```
>>> def f1(a):
...     a[0] = None
...
>>> mylist = [1, 2, 3]; f1(mylist); print(mylist)
[None, 2, 3]
```