Capitolo 3

Funzioni

Nell'ambito della programmazione, una **funzione** è una serie di istruzioni che esegue un calcolo, alla quale viene assegnato un nome. Per definire una funzione, dovete specificarne il nome e scrivere la serie di istruzioni. In un secondo tempo, potete "chiamare" la funzione mediante il nome che le avete assegnato.

3.1 Chiamate di funzione

Abbiamo già visto un esempio di una **chiamata di funzione**:

```
>>> type(42)
<class 'int'>
```

Il nome di questa funzione è `type`. L'espressione tra parentesi è chiamata **argomento** della funzione, e il risultato che produce è il tipo di valore dell'argomento che abbiamo inserito.

Si usa dire che una funzione "prende" o "riceve" un argomento e, una volta eseguita l'elaborazione, "ritorna" o "restituisce" un risultato. Il risultato è detto **valore di ritorno**.

Python contiene una raccolta di funzioni per convertire i valori da un tipo a un altro. La funzione `int` prende un dato valore e lo converte, se possibile, in un numero intero. Se la conversione è impossibile, Python comunica che si è verificato un errore:

```
>>> int('32')
32
>>> int('Ciao')
ValueError: invalid literal for int(): Ciao
```

`int` può anche convertire valori in virgola mobile in interi, ma non arrotonda bensì tronca la parte decimale.

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

La funzione `float` converte interi e stringhe in numeri a virgola mobile:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Infine, `str` converte l'argomento in una stringa:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2 Funzioni matematiche

Python è provvisto di un modulo matematico che comprende buona parte delle funzioni matematiche d'uso frequente. Un **modulo** è un file che contiene una raccolta di funzioni correlate.

Prima di poter usare le funzioni contenute in un modulo, lo dobbiamo importare con un'**istruzione di importazione**:

```
>>> import math
```

Questa istruzione crea un **oggetto modulo** chiamato math. Se visualizzate l'oggetto modulo, ottenete alcune informazioni a riguardo:

```
>>> math
<module 'math' (built-in)>
```

L'oggetto modulo contiene le funzioni e le variabili definite all'interno del modulo stesso. Per accedere a una funzione del modulo, dovete specificare, nell'ordine, il nome del modulo e il nome della funzione, separati da un punto. Questo formato è chiamato **notazione a punto** o *dot notation*.

```
>>> rapporto = potenza_segnale / potenza_rumore
>>> decibel = 10 * math.log10(rapporto)

>>> radianti = 0.7
>>> altezza = math.sin(radianti)
```

Il primo esempio utilizza la funzione `math.log10` per calcolare un rapporto segnale/rumore in decibel (a condizione che siano stati definiti i valori di `potenza_segnale` e `potenza_rumore`). Il modulo math contiene anche `log`, che calcola i logaritmi naturali in base `e`.

Il secondo esempio calcola il seno della variabile `radianti`. Il nome della variabile spiega già che `sin` e le altre funzioni trigonometriche (`cos`, `tan`, ecc.) accettano argomenti espressi in radianti. Per convertire da gradi in radianti occorre dividere per 180 e moltiplicare per π:

```
>>> gradi = 45
>>> radianti = gradi / 180.0 * math.pi
>>> math.sin(radianti)
0.707106781187
```

L'espressione `math.pi` ricava la variabile `pi` dal modulo matematico. Il suo valore è un numero decimale, approssimazione di π, accurata a circa 15 cifre.

Se ricordate la trigonometria, potete verificare il risultato precedente confrontandolo con la radice quadrata di 2 diviso 2:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.3 Composizione

Finora, abbiamo considerato gli elementi di un programma - variabili, espressioni e istruzioni - separatamente, senza discutere di come utilizzarli insieme.

Una delle caratteristiche più utili dei linguaggi di programmazione è la loro capacità di prendere dei piccoli mattoni e **comporli** tra loro. Per esempio, l'argomento di una funzione può essere un qualunque tipo di espressione, operazioni aritmetiche incluse:

```
x = math.sin(gradi / 360.0 * 2 * math.pi)
```

E anche chiamate di funzione:

```
x = math.exp(math.log(x+1))
```

In linea generale, dovunque potete mettere un valore potete anche mettere un'espressione a piacere, con un'eccezione: il lato sinistro di un'istruzione di assegnazione deve essere un nome di variabile. Ogni altra espressione darebbe un errore di sintassi (vedremo più avanti le eccezioni a questa regola).

```
>>> minuti = ore * 60                # giusto
>>> ore * 60 = minuti                # sbagliato!
SyntaxError: can't assign to operator
```

3.4 Aggiungere nuove funzioni

Finora abbiamo usato solo funzioni predefinite o "built-in", che sono parte integrante di Python, ma è anche possibile crearne di nuove. Una **definizione di funzione** specifica il nome di una nuova funzione e la serie di istruzioni che viene eseguita quando la funzione viene chiamata.

Ecco un esempio:

```
def stampa_brani():
    print('Terror di tutta la foresta egli è,')
    print("Con l'ascia in mano si sente un re.")
```

`def` è una parola chiave riservata che indica la definizione di una nuova funzione. Il nome della funzione è `stampa_brani`. Le regole per i nomi delle funzioni sono le stesse dei nomi delle variabili: lettere, numeri e underscore (`_`) sono permessi, ma il primo carattere non può essere un numero. Non si possono usare parole riservate, e bisogna evitare di avere una funzione e una variabile con lo stesso nome

Le parentesi vuote dopo il nome indicano che la funzione non accetta alcun argomento.

La prima riga della definizione di funzione è chiamata **intestazione**; il resto è detto **corpo**. L'intestazione deve terminare con i due punti, e il corpo deve essere obbligatoriamente indentato, cioè deve avere un rientro rispetto all'intestazione. Per convenzione, l'indentazione è sempre di quattro spazi. Il corpo può contenere un qualsiasi numero di istruzioni.

Le stringhe nelle istruzioni di stampa sono racchiuse tra apici (`' '`) oppure virgolette (`" "`). Virgolette e apici sono equivalenti; la maggioranza degli utenti usa gli apici, eccetto nei casi in cui nel testo da stampare sono contenuti degli apici (che possono essere usati anche come apostrofi o accenti). In questi casi, frequenti con l'italiano, bisogna usare le virgolette.

Virgolette e apici devono essere alti e di tipo indifferenziato, quelli che trovate tra i simboli in alto sulla vostra tastiera. Altre virgolette "tipografiche", come quelle in questa frase, non sono valide in Python.

Se scrivete una funzione in modalità interattiva, l'interprete mette tre puntini di sospensione (...) per indicare che la definizione non è completa:

```
>>> def stampa_brani():
...     print('Terror di tutta la foresta egli è,')
...     print("Con l'ascia in mano si sente un re.")
...
```

Per concludere la funzione, dovete inserire una riga vuota.

La definizione di una funzione crea un **oggetto funzione** che è di tipo `function`:

```
>>> print(stampa_brani)
<function stampa_brani at 0xb7e99e9c>
>>> type(stampa_brani)
<class 'function'>
```

La sintassi per chiamare la nuova funzione è la stessa che abbiamo visto per le funzioni predefinite:

```
>>> stampa_brani()
Terror di tutta la foresta egli è,
Con l'ascia in mano si sente un re.
```

Una volta definita una funzione, si può utilizzarla all'interno di un'altra funzione. Per esempio, per ripetere due volte il brano precedente possiamo scrivere una funzione `ripeti_brani`:

```
def ripeti_brani():
    stampa_brani()
    stampa_brani()
```

E quindi chiamare `ripeti_brani`:

```
>>> ripeti_brani()
Terror di tutta la foresta egli è,
Con l'ascia in mano si sente un re.
Terror di tutta la foresta egli è,
Con l'ascia in mano si sente un re.
```

Ma a dire il vero, la canzone del taglialegna non fa così!

3.5 Definizioni e loro utilizzo

Raggruppando assieme i frammenti di codice del Paragrafo precedente, il programma diventa:

```
def stampa_brani():
    print('Terror di tutta la foresta egli è,')
    print("Con l'ascia in mano si sente un re.")

def ripeti_brani():
    stampa_brani()
    stampa_brani()

ripeti_brani()
```

Questo programma contiene due definizioni di funzione: `stampa_brani` e `ripeti_brani`. Le definizioni di funzione sono eseguite come le altre istruzioni, ma il loro effetto è solo quello di creare una nuova funzione. Le istruzioni all'interno di una definizione non vengono eseguite fino a quando la funzione non viene chiamata, e la definizione di per sé non genera alcun risultato.

Ovviamente, una funzione deve essere definita prima di poterla usare: la definizione della funzione deve sempre precedere la sua chiamata.

Come esercizio, spostate l'ultima riga del programma all'inizio, per fare in modo che la chiamata della funzione appaia prima della definizione. Eseguite il programma e guardate che tipo di messaggio d'errore ottenete.

Ora riportate la chiamata della funzione al suo posto, e spostate la definizione di `stampa_brani` dopo la definizione di `ripeti_brani`. Cosa succede quando avviate il programma?

3.6 Flusso di esecuzione

Per essere sicuri che una funzione sia stata definita prima di essere utilizzata, dovete conoscere l'ordine in cui le istruzioni vengono eseguite, che è chiamato **flusso di esecuzione**.

L'esecuzione inizia sempre dalla prima istruzione del programma; quindi, le istruzioni successive sono eseguite una alla volta, procedendo dall'alto verso il basso.

Le definizioni di funzione non cambiano il flusso di esecuzione del programma, ma ricordate che le istruzioni all'interno delle funzioni non vengono eseguite fino a quando la funzione non viene chiamata.

Quando viene chiamata una funzione, si genera una specie di deviazione nel flusso di esecuzione: anziché proseguire con l'istruzione successiva, il flusso salta nel corpo della funzione chiamata, ne esegue le istruzioni, e infine riprende il percorso dal punto che aveva lasciato.

Parrebbe tutto abbastanza semplice, se non fosse che una funzione può chiamarne un'altra. Mentre si trova all'interno di una funzione, il programma può dover eseguire le istruzioni che si trovano in un'altra funzione. Poi, mentre esegue quella nuova funzione, il programma potrebbe andare ad eseguirne un'altra ancora!

Fortunatamente, Python sa tener bene traccia di dove si trova: ogni volta che una funzione viene completata, il programma ritorna al punto della funzione chiamante che aveva lasciato. E una volta giunto alla fine, termina il suo lavoro.

Concludendo, nel leggere un programma non è sempre opportuno farlo dall'alto in basso. Spesso è più logico seguire il flusso di esecuzione.

3.7 Parametri e argomenti

Alcune delle funzioni che abbiamo visto richiedono degli argomenti. Per esempio, se volete trovare il seno di un numero chiamando la funzione `math.sin`, dovete passarle quel numero come argomento. Alcune funzioni ricevono più di un argomento: a `math.pow` ne servono due, che sono la base e l'esponente dell'operazione di elevamento a potenza.

All'interno della funzione, gli argomenti che le vengono passati sono assegnati ad altrettante variabili chiamate **parametri**. Ecco un esempio di definizione di una funzione che riceve un argomento:

```
def stampa2volte(bruce):
    print(bruce)
    print(bruce)
```

Questa funzione assegna l'argomento ricevuto ad un parametro chiamato `bruce`. Quando la funzione viene chiamata, stampa il valore del parametro (qualunque esso sia) due volte.

Questa funzione elabora qualunque valore che possa essere stampato.

```
>>> stampa2volte('Spam')
Spam
Spam
>>> stampa2volte(42)
42
42
>>> stampa2volte(math.pi)
3.14159265359
3.14159265359
```

Le stesse regole di composizione che valgono per le funzioni predefinite si applicano anche alle funzioni definite da un programmatore, pertanto possiamo usare come argomento per `stampa2volte` qualsiasi espressione:

```
>>> stampa2volte('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> stampa2volte(math.cos(math.pi))
-1.0
-1.0
```

L'argomento viene valutato prima della chiamata alla funzione, pertanto nell'esempio appena proposto le espressioni `'Spam '*4` e `math.cos(math.pi)` vengono valutate una volta sola.

Potete anche usare una variabile come argomento di una funzione:

```
>>> michael = 'Eric, the half a bee.'
>>> stampa2volte(michael)
Eric, the half a bee.
Eric, the half a bee.
```

Il nome della variabile che passiamo come argomento (`michael`) non ha niente a che fare con il nome del parametro nella definizione della funzione (`bruce`). Non ha importanza come era stato denominato il valore di partenza (nel codice chiamante); qui in `stampa2volte`, chiamiamo tutto quanto `bruce`.

3.8 Variabili e parametri sono locali

Quando create una variabile in una funzione, essa è **locale**, cioè esiste solo all'interno della funzione. Per esempio:

```
def cat2volte(parte1, parte2):
    cat = parte1 + parte2
    stampa2volte(cat)
```

Questa funzione prende due argomenti, li concatena e poi stampa il risultato per due volte. Ecco un esempio che la utilizza:

```
>>> riga1 = 'Bing tiddle '
>>> riga2 = 'tiddle bang.'
>>> cat2volte(riga1, riga2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

Quando `cat2volte` termina, la variabile `cat` viene distrutta. Se provassimo a stamparla, otterremmo infatti un messaggio d'errore:

```
>>> print(cat)
NameError: name 'cat' is not defined
```

Anche i parametri sono locali: esternamente alla funzione `stampa2volte`, non esiste nulla di nome `bruce`.

3.9 Diagrammi di stack

Per tenere traccia di quali variabili possono essere usate e dove, a volte può risultare utile disegnare un **diagramma di stack**. Come i diagrammi di stato, i diagrammi di stack mostrano il valore di ciascuna variabile, ma in più indicano a quale funzione essa appartiene.

Ciascuna funzione è rappresentata da un **frame**, un riquadro che riporta a fianco il nome della funzione e all'interno un elenco dei suoi parametri e delle sue variabili. Nel caso dell'esempio precedente, il diagramma di stack è illustrato in Figura 3.1.

I frame sono disposti in una pila che indica quale funzione ne ha chiamata un'altra e così via. Nell'esempio, `stampa2volte` è stata chiamata da `cat2volte`, e `cat2volte` è stata a sua volta chiamata da `__main__`, che è un nome speciale per il frame principale. Quando si crea una variabile che è esterna ad ogni funzione, essa appartiene a `__main__`.

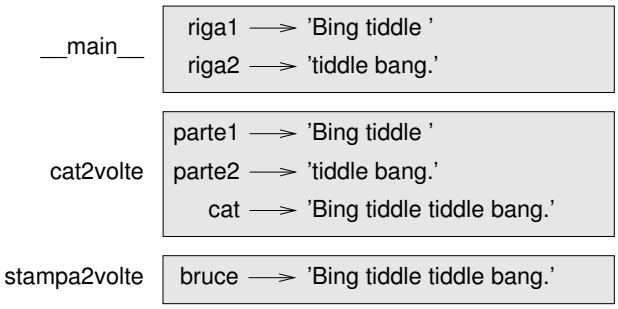

Figura 3.1: Diagramma di stack.

Ogni parametro fa riferimento allo stesso valore del suo argomento corrispondente. Così, `parte1` ha lo stesso valore di `riga1`, `parte2` ha lo stesso valore di `riga2`, e `bruce` ha lo stesso valore di `cat`.

Se si verifica un errore durante la chiamata di una funzione, Python mostra il nome della funzione, il nome della funzione che l'ha chiamata, il nome della funzione che a sua volta ha chiamato quest'ultima e così via, fino a raggiungere il primo livello che è sempre `__main__`.

Ad esempio se cercate di accedere a `cat` dall'interno di `stampa2volte`, ottenete un errore di tipo `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat2volte(riga1, riga2)
  File "test.py", line 5, in cat2volte
    stampa2volte(cat)
  File "test.py", line 9, in stampa2volte
    print(cat)
NameError: name 'cat' is not defined
```

Questo elenco di funzioni è detto **traceback**. Il traceback vi dice in quale file è avvenuto l'errore, e in quale riga, e quale funzione era in esecuzione in quel momento. Mostra anche la riga di codice che ha causato l'errore.

L'ordine delle funzioni nel traceback è lo stesso di quello dei frame nel diagramma di stack. La funzione attualmente in esecuzione si trova in fondo all'elenco.

3.10 Funzioni "produttive" e funzioni "vuote"

Alcune delle funzioni che abbiamo usato, come le funzioni matematiche, restituiscono dei risultati; in mancanza di definizioni migliori, personalmente le chiamo **funzioni "produttive"**. Altre funzioni, come `stampa2volte`, eseguono un'azione ma non restituiscono alcun valore. Le chiameremo **funzioni "vuote"**.

Quando chiamate una funzione produttiva, quasi sempre è per fare qualcosa di utile con il suo risultato, tipo assegnarlo a una variabile o usarlo come parte di un'espressione.

```
x = math.cos(radianti)
aureo = (math.sqrt(5) + 1) / 2
```

Se chiamate una funzione in modalità interattiva, Python ne mostra il risultato:

```
>>> math.sqrt(5)
2.2360679774997898
```

Ma in uno script, se chiamate una funzione produttiva così come è, il valore di ritorno è perso!

```
math.sqrt(5)
```

Questo script in effetti calcola la radice quadrata di 5, ma non conserva nè visualizza il risultato, per cui non è di grande utilità.

Le funzioni vuote possono visualizzare qualcosa sullo schermo o avere qualche altro effetto, ma non restituiscono un valore. Se provate comunque ad assegnare il risultato ad una variabile, ottenete un valore speciale chiamato `None` (nulla).

```
>>> risultato = stampa2volte('Bing')
Bing
Bing
>>> print(risultato)
None
```

Il valore `None` non è la stessa cosa della stringa `'None'`. È un valore speciale che appartiene ad un tipo tutto suo:

```
>>> type(None)
<class 'NoneType'>
```

Le funzioni che abbiamo scritto finora, sono tutte vuote. Cominceremo a scriverne di produttive tra alcuni capitoli.

3.11 Perché usare le funzioni?

Potrebbe non esservi ancora ben chiaro perché valga la pena di suddividere il programma in funzioni. Ecco alcuni motivi:

- Creare una nuova funzione vi dà modo di dare un nome a un gruppo di istruzioni, rendendo il programma più facile da leggere e da correggere.
- Le funzioni possono rendere un programma più breve, eliminando il codice ripetitivo. Se in un secondo tempo dovete fare una modifica, basterà farla in un posto solo.
- Dividere un programma lungo in funzioni vi permette di correggere le parti una per una, per poi assemblarle in un complesso funzionante.
- Funzioni ben fatte sono spesso utili per più programmi. Quando ne avete scritta e corretta una, la potete riutilizzare tale e quale.

3.12 Debug

Saper rintracciare e correggere gli errori è una essenziale qualità che dovete acquisire. Anche se a volte può essere demotivante, si tratta infatti di una delle parti più intellettualmente ricche, stimolanti ed interessanti della programmazione.

Possiamo paragonare il debug al lavoro di un investigatore: avete a disposizione degli indizi e dovete ricostruire quali processi ed eventi hanno prodotto il risultato che osservate.

Il debug è anche simile ad una scienza sperimentale. Quando pensate di aver capito cosa può avere provocato un errore, modificate il programma di conseguenza e riprovate di nuovo. Se l'ipotesi era giusta, avete saputo prevedere il risultato della modifica e vi siete avvicinati di un passo ad un programma funzionante. Se l'ipotesi era sbagliata, ne dovete formulare un'altra. Come disse Sherlock Holmes: "Una volta eliminato l'impossibile, qualsiasi cosa rimanga, per quanto improbabile, deve essere la verità." (A. Conan Doyle, *Il segno dei quattro*)

Per alcuni, la programmazione e la rimozione degli errori sono in fondo la stessa cosa: programmare è una procedura di graduale rimozione degli errori da un programma, fino a quando non funziona a dovere. L'idea di fondo è di iniziare con un programma funzionante e di fare ogni volta piccole modifiche, effettuandone man mano il debug.

Linux, ad esempio, è un sistema operativo fatto da milioni di righe di codice, ma nacque come un semplice programma che Linus Torvalds usava per esplorare il chip Intel 80386. Secondo Larry Greenfields, "Uno dei progetti iniziali di Linus era un programma che doveva visualizzare alternativamente una sequenza di AAAA e BBBB. Questo programma si è poi evoluto in Linux". (*The Linux Users' Guide* Beta Version 1).

3.13 Glossario

funzione: Una serie di istruzioni dotata di un nome che esegue una certa operazione utile. Le funzioni possono o meno ricevere argomenti e possono o meno produrre un risultato.

definizione di funzione: Istruzione che crea una nuova funzione, specificandone il nome, i parametri, e le istruzioni che contiene.

oggetto funzione: Valore creato da una definizione di funzione. Il nome della funzione è una variabile che fa riferimento a un oggetto funzione.

intestazione: La prima riga di una definizione di funzione.

corpo: La serie di istruzioni all'interno di una definizione di funzione.

parametro: Un nome usato all'interno di una funzione che fa riferimento al valore passato come argomento.

chiamata di funzione: Istruzione che esegue una funzione. Consiste nel nome della funzione seguito da un elenco di argomenti tra parentesi.

argomento: Un valore fornito (passato) a una funzione quando viene chiamata. Questo valore viene assegnato al corrispondente parametro nella funzione.

variabile locale: Variabile definita all'interno di una funzione e che puo essere usata solo all'interno della funzione.

valore di ritorno: Il risultato di una funzione. Se una chiamata di funzione viene usata come espressione, il valore di ritorno è il valore dell'espressione.

funzione "produttiva": Una funzione che restituisce un valore.

funzione "vuota": Una funzione che restituisce sempre `None`.

`None`**:** Valore speciale restituito dalle funzioni vuote.

modulo: Un file che contiene una raccolta di funzioni correlate e altre definizioni.

istruzione import: Istruzione che legge un file modulo e crea un oggetto modulo utilizzabile.

oggetto modulo: Valore creato da un'istruzione `import` che fornisce l'accesso ai valori definiti in un modulo.

dot notation o notazione a punto: Sintassi per chiamare una funzione di un modulo diverso, specificando il nome del modulo seguito da un punto e dal nome della funzione.

composizione: Utilizzare un'espressione come parte di un'espressione più grande o un'istruzione come parte di un'istruzione più grande.

flusso di esecuzione: L'ordine in cui vengono eseguite le istruzioni nel corso di un programma.

diagramma di stack: Rappresentazione grafica di una serie di funzioni impilate, delle loro variabili e dei valori a cui fanno riferimento.

frame: Un riquadro in un diagramma di stack che rappresenta una chiamata di funzione. Contiene le variabili locali e i parametri della funzione.

traceback: Elenco delle funzioni in corso di esecuzione, visualizzato quando si verifica un errore.

3.14 Esercizi

Esercizio 3.1. *Scrivete una funzione chiamata* `giustif_destra` *che richieda una stringa* `s` *come parametro e stampi la stringa con tanti spazi iniziali da far sì che l'ultima lettera della stringa cada nella colonna 70 del display.*

```
>>> giustif_destra('monty')
                                                                monty
```

Suggerimento: usate concatenamento delle stringhe e ripetizione. Inoltre, Python contiene una funzione predefinita chiamata `len` *che restituisce la lunghezza di una stringa, ad esempio il valore di* `len('monty')` *è 5.*

Esercizio 3.2. *Un oggetto funzione è un valore che potete assegnare a una variabile o passare come argomento. Ad esempio,* `fai2volte` *è una funzione che accetta un oggetto funzione come argomento e la chiama per due volte.*

```
def fai2volte(f):
    f()
    f()
```

Ecco un esempio che usa `fai2volte` *per chiamare una funzione di nome* `stampa_spam` *due volte.*

```
def stampa_spam():
    print('spam')

fai2volte(stampa_spam)
```

1. *Scrivete questo esempio in uno script e provatelo.*
2. *Modificate* `fai2volte` *in modo che accetti due argomenti, un oggetto funzione e un valore, e che chiami la funzione due volte passando il valore come argomento.*
3. *Copiate nel vostro script la definizione di* `stampa_2volte` *che abbiamo visto nel corso di questo capitolo.*
4. *Usate la versione modificata di* `fai2volte` *per chiamare* `stampa_2volte` *per due volte, passando* `'spam'` *come argomento.*
5. *Definite una nuova funzione di nome* `fai_quattro` *che richieda un oggetto funzione e un valore e chiami la funzione per 4 volte, passando il valore come argomento. Dovrebbero esserci solo due istruzioni nel corpo di questa funzione, non quattro.*

Soluzione: `http://thinkpython2.com/code/do_four.py`.

Esercizio 3.3. *Nota: questo esercizio dovrebbe essere svolto con le sole istruzioni e caratteristiche del linguaggio imparate finora.*

1. *Scrivete una funzione che disegni una griglia come questa:*

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

   Suggerimento: per stampare più di un valore per riga, stampate una sequenza di valori separati da virgole:

```
print('+', '-')
```

   Di default, `print` *va a capo; si può però variare questo comportamento e restare sulla stessa riga, inserendo uno spazio, in questo modo:*

```
print('+', end=' ')
print('-')
```

   L'output di queste istruzioni è `'+ -'`.

   Una funzione `print` *priva di argomento, termina la riga e va a capo.*

2. *Scrivete una funzione che disegni una griglia simile, con quattro righe e quattro colonne.*

Soluzione: `http://thinkpython2.com/code/grid.py`. *Fonte: Esercizio tratto da Oualline,* Practical C Programming, Third Edition*, O'Reilly Media, 1997.*