- Marco Buttu -

PROGRAMMARECON

Python

Guida completa

Guida completa al linguaggio, aggiornata allo stato dell'arte >>

Applicazioni, ambienti virtuali e docstring validation testing >>

Programmazione a oggetti e decoratori >>

Metaclassi, descriptor e TDD >>

*pro Digital Life Style

PROGRAMMARECON

Python

Guida completa

Marco Buttu

LSWR

Phyton | Guida completa

L'autore: Marco Buttu

Collana:

PROGRAMMARECON

Publisher: Fabrizio Comolli
Editor: Marco Aleotti
Progetto grafico e impaginazione: Roberta Venturieri
Immagine di copertina: © Hlubokidzianis | Dreamstime.com

ISBN: 978-88-6895-024-8

Finito di stampare nel mese di febbraio 2014 presso "Press Grafica" s.r.l., Gravellona Toce (VB)

Sommario

Introduzione

Python è un linguaggio di programmazione multipiattaforma, robusto e maturo, a cui si affidano con successo le più prestigiose aziende e organizzazioni a livello mondiale, come Google, YouTube, Intel, Yahoo! e la NASA. I suoi ambiti di utilizzo sono molteplici: applicazioni web, giochi e multimedia, interfacce grafiche, networking, applicazioni scientifiche, intelligenza artificiale, programmazione di sistema e tanto altro ancora.
L'obiettivo di questo libro è insegnare a programmare con Python, nel modo giusto (The Pythonic Way). Il tema centrale è, quindi, il linguaggio, in tutti i suoi aspetti, che viene affrontato in modo dettagliato sia dal punto di vista teorico sia da quello pratico. Il libro è aggiornato alla versione di Python 3.4, rilasciata nel 2014.

Il pubblico a cui si rivolge

Il libro si rivolge sia a chi intende imparare a programmare con Python, sia a chi già conosce il linguaggio ma vuole approfondire gli argomenti più avanzati, come, ad esempio, i decoratori, le metaclassi e i descriptor.
La lettura sarà probabilmente più agevole per chi ha precedenti esperienze di programmazione, ma il libro è alla portata di tutti, perché nulla è dato per scontato. Si parte, infatti, dallo studio delle basi del linguaggio e si arriva, seguendo un percorso graduale costruito attorno a una ricca serie di esempi ed esercizi, agli argomenti più avanzati.
Questo libro non è solamente una guida a Python, ma anche un manuale di programmazione, poiché vengono affrontate numerose tematiche di carattere generale, come, ad esempio, l'aritmetica del calcolatore e le problematiche a essa connesse, lo standard Unicode e il test driven development.

I contenuti

Il libro è suddiviso in sei capitoli e tre appendici. Il **primo** è un sommario dell'intero libro, poiché introduce, in modo graduale, numerosi argomenti, come gli oggetti built-in, i moduli, i file, le funzioni, le classi e la libreria standard. Al termine di questo capitolo il lettore dovrebbe essere già produttivo e in grado di scrivere dei programmi Python non banali.

Nei capitoli che vanno **dal secondo al quinto** vengono approfonditi e integrati tutti gli argomenti visti nel primo capitolo: il secondo capitolo è centrato sul core datatype, il terzo sulle funzioni, sui generatori e sui file, il quarto sui moduli, sull'installazione e sulla distribuzione delle applicazioni, il quinto sulla programmazione orientata agli oggetti.

Il **sesto** capitolo è dedicato ad argomenti avanzati: modello a oggetti di Python, metaclassi, attributi magici, descriptor e test driven development. La trattazione sarà graduale, ma allo stesso tempo molto dettagliata.

Non vi è un capitolo del libro appositamente dedicato alla libreria standard, poiché questo argomento è molto vasto e non basterebbe un libro intero per trattarlo in modo completo. Si è preferito seguire un approccio pratico, discutendo dei vari moduli di interesse al momento opportuno, a mano a mano che se ne presenta l'occasione. Questo consente sia di fare pratica con i moduli più comunemente utilizzati, sia di familiarizzare con la documentazione online.

Al termine di ogni capitolo è presente un **esercizio conclusivo**, che ha lo scopo non solo di analizzare dei programmi completi, ma soprattutto di affrontare altre importanti tematiche e di esplorare la libreria standard. In questi esercizi si vedrà, infatti, come fare il parsing degli argomenti da linea di comando, realizzare programmi portabili che possono essere eseguiti allo stesso modo su tutti i sistemi operativi, lavorare con le date, utilizzare le wildcard e le espressioni regolari, effettuare il test driven development e tanto altro ancora.

Ogni esercizio inizia mostrando il codice – che dopo una prima lettura probabilmente sembrerà incomprensibile – e prosegue con la spiegazione del significato di ogni linea, in modo che, al termine, tutti gli aspetti risultino chiari per il lettore.

La parte finale del libro è costituita da tre **appendici**. L'appendice **A** descrive i comandi Unix-like utilizzati nel libro, l'appendice **B** i principali punti di rottura tra Python 2 e Python 3, l'appendice **C** il meccanismo di buffering dei file.

Materiale e contatti

Il codice sorgente degli esercizi conclusivi e l'errata corrige sono disponibili via Internet, all'indirizzo:

http://code.google.com/p/the-pythonic-way/

L'autore è ben lieto di esser contattato per fornire dei chiarimenti, o anche, più semplicemente, per avere uno scambio di opinioni. Lo si può fare scrivendo direttamente al suo indirizzo E-mail:

marco.buttu@gmail.com

indicando nel soggetto il testo *Python - Guida completa*.

Ringraziamenti

L'autore dedica questo libro a Micky, sua insostituibile metà, ringraziandola per la pazienza e per la comprensione. Senza il suo aiuto non sarebbe stato possibile scrivere questo libro.
Un doveroso ringraziamento spetta, inoltre, a tutti coloro che hanno speso del tempo per scambiare con l'autore delle opinioni in merito a Python, in particolare a Steven D'Aprano, per il suo contributo sul gruppo di discussione **comp.lang.python** e sulla mailing list **python-ideas**.
L'autore ringrazia, inoltre, Andrea Saba, Franco Buffa e Marco Bartolini per gli utili consigli.

1

I fondamenti del linguaggio

In questo capitolo costruiremo solide **fondamenta** pratico-teoriche che consentiranno di essere **subito produttivi**. Introdurremo gradualmente numerosi argomenti: oggetti **built-in**, **moduli**, **file**, **funzioni**, **classi** e **libreria standard**. L'obiettivo è ambizioso: fornire le basi a chi intende **imparare a programmare** nel modo corretto con Python e, allo stesso tempo, offrire spunti interessanti ai **programmatori Python esperti**.

Introduzione a Python

Python nasce nel dicembre del 1989 per opera dell'informatico olandese Guido van Rossum.

Dopo aver lavorato per quattro anni (dal 1982 al 1986) allo sviluppo del linguaggio di programmazione ABC, presso il Centrum voor Wiskunde en Informatica (CWI) di Amsterdam, dal 1986 Guido inizia a collaborare allo sviluppo di Amoeba, un sistema operativo distribuito nato anch'esso ad Amsterdam (1981), alla Vrije Universiteit. Alla fine degli anni Ottanta il gruppo si rende conto che Amoeba necessita di un linguaggio di scripting, cosicché Guido, mentre si trova a casa per le vacanze di Natale del 1989, un po' per hobby e un po' per contribuire allo sviluppo di Amoeba, decide di avviare un suo progetto personale.

Cerca di fare mente locale su quanto ha appreso durante il periodo di lavoro su ABC. Quell'esperienza è stata piuttosto frustrante, ma alcune caratteristiche di ABC gli piacciono, tanto da pensare di usarle come fondamenti del suo nuovo linguaggio:

- l'indentazione per indicare i blocchi di istruzioni annidate;
- nessuna dichiarazione delle variabili;
- stringhe e liste di lunghezza arbitraria.

Su queste basi inizia a scrivere in C un interprete per il suo futuro linguaggio di programmazione, che battezza con il nome di *Python* in onore della sua serie televisiva preferita: *Monty Python's Flying Circus*.
Nel 1990 Guido termina la prima implementazione dell'interprete, che rilascia a uso interno alla CWI. Nel febbraio del 1991 rende pubblico il codice su **alt.sources**, indicando la versione come la numero 0.9.0. Nel 1994 viene creato **comp.lang.python**, il primo gruppo di discussione su Python, e l'anno successivo nasce il sito ufficiale www.python.org.

Sviluppo di Python

Python è sviluppato, mantenuto e rilasciato da un gruppo di persone coordinato da Guido van Rossum, il quale ha l'ultima parola su tutte le decisioni relative sia al linguaggio sia alla libreria standard. Per questo motivo nel 1995 è stato dato a Guido il titolo di Benevolent Dictator For Life (BDFL). Il 6 marzo 2001 viene fondata la Python Software Foundation (PSF), un'organizzazione not-for-profit che detiene il diritto d'autore su Python e ne promuove la diffusione. La PSF è presieduta da Guido van Rossum e annovera tra i suoi membri il cuore degli sviluppatori di Python più tante altre personalità nominate in virtù del loro notevole contributo al linguaggio.
Le *major version* (le versioni principali, quelle che si differenziano per il primo numero della versione, detto *major number*) vengono rilasciate a distanza di diversi anni l'una dall'altra. La 1.0 è stata rilasciata nel 1994, la 2.0 nel 2000 e la 3.0 nel 2008. Le *minor version* (le versioni minori, quelle che hanno lo stesso major number e si differenziano per il primo numero dopo il punto), invece, vengono rilasciate ogni uno o due anni.
Le funzionalità che vengono aggiunte nelle minor version sono retrocompatibili, nel senso che tutto il codice scritto per una minor version funzionerà nella stessa maniera anche con le minor version successive. Quindi il codice scritto per la versione *3.x* funzionerà in modo identico anche con tutte le versioni 3.y, con y maggiore di x.
Per quanto riguarda le major version, invece, non è garantita la retrocompatibilità. Quando del codice di una major version x non può essere eseguito con una major version y, con x < y, oppure può essere eseguito ma dà luogo a un diverso risultato, si dice che è incompatibile con la versione y.

NOTA

Questo libro è aggiornato a Python 3.4. Il codice è stato eseguito sia con Python 3.3 sia con Python 3.4 (le funzionalità introdotte con la 3.4, ovviamente, sono state testate solo con la 3.4). Per conoscere le principali incompatibilità tra Python 2 e Python 3 si rimanda all'Appendice B, dal titolo *Principali punti di rottura tra Python 2 e Python 3*.

Per indicare le versioni nelle quali vengono risolti una serie di bug presenti nella minor version, viene utilizzato un terzo numero, chiamato *micro number*, o anche *patch number*. Ad esempio, nella versione x.y.1 di Python il numero 1 indica il micro number. Quindi la 3.4.1 è la prima *bug-fix release* di Python 3.4.
Infine, può capitare di vedere un codice in coda al numero di versione, come ad esempio 3.4.1a3, 3.4.1b2, 3.4.1c4. Questo codice è usato per indicare le *sub-release*. I codici *a1*, *a2*, ... *aN* indicano le *alpha release*, le quali possono aggiungere nuove funzionalità (ad esempio, nella 3.4.1a2 potrebbero essere presenti funzionalità non comprese nella 3.4.1a1). I codici *b1*, *b2*, ... *bN* indicano le *beta release*, le quali possono solo risolvere i bug e non possono aggiungere nuove funzionalità. I codici *c1*, *c2*, ... *cN* indicano le release *candidate*, nelle quali il bug-fix viene scrupolosamente verificato dal core development.
Lo strumento utilizzato per proporre i cambiamenti al linguaggio è la *Python Enhancement Proposal*, indicata con l'acronimo PEP. Le PEP sono documenti pubblici che vengono discussi dagli sviluppatori di Python e dalla comunità, per poi essere approvati o respinti da Guido. Le PEP riguardano vari aspetti del linguaggio e sono identificate da un codice unico (per esempio, PEP-0008). L'archivio di tutte le PEP si trova alla pagina http://www.python.org/dev/peps/. Per raggiungere la pagina di una specifica PEP dobbiamo aggiungere */pep-code/* all'indirizzo dell'archivio; per esempio, la pagina della PEP-0008 è raggiungibile all'URL http://www.python.org/dev/peps/pep-0008/. Teniamo questo a mente, poiché faremo ampio riferimento alle PEP nel corso del libro.

NOTA

La nomenclatura e la gestione del rilascio delle versioni di Python è discussa nella PEP-0101, dal titolo *Doing Python Releases 101*, mentre le micro release e le sub-release sono discusse nella PEP-0102.

Lo stato dell'arte

Python è un linguaggio robusto e maturo, utilizzato in tantissimi ambiti: web, sviluppo di interfacce grafiche, programmazione di sistema, networking, database, calcolo

numerico e applicazioni scientifiche, programmazione di giochi e multimedia, grafica, intelligenza artificiale e tanto altro ancora.
È un linguaggio multipiattaforma, ovvero disponibile per tutti i principali sistemi operativi, ed è automaticamente incluso nelle distribuzioni Linux e nei computer Macintosh. Inoltre fornisce tutti gli strumenti per scrivere in modo semplice programmi portabili, ovvero che si comportano alla stessa maniera se eseguiti su differenti piattaforme. Vedremo un esempio eloquente di codice portabile nell'esercizio conclusivo al termine di questo capitolo.
È utilizzato con successo in tutto il mondo da svariate aziende e organizzazioni, tra le quali Google, la NASA, YouTube, Intel, Yahoo! Groups, reddit, Spotify Ltd, OpenStack e Dropbox Inc. Quest'ultima merita una nota a parte, poiché la sua storia ci consente di evidenziare diversi punti di forza di Python.
Dropbox è un software multipiattaforma che offre un servizio di file hosting e sincronizzazione automatica dei file tramite web. La sua prima versione è stata rilasciata nel settembre del 2008 e in pochissimo tempo ha avuto un successo sorprendente, raggiungendo i 50 milioni di utenti nell'ottobre del 2011 e i 100 milioni l'anno successivo, come annunciato il 12 novembre del 2012 da Drew Houston, uno dei due fondatori della Dropbox Inc.
Dopo nemmeno un mese dall'annuncio di questo incredibile risultato, il 7 dicembre 2012 Drew stupisce tutti con un'altra clamorosa notizia. Guido van Rossum, dopo aver contribuito per sette anni alle fortune di Google, si unisce al team di Dropbox:

Oggi siamo entusiasti di dare il benvenuto, in circostanze insolite, a un nuovo membro della famiglia Dropbox. Sebbene si unisca a noi solo ora, i suoi contributi a Dropbox risalgono al primo giorno, sin dalla primissima linea di codice.
Solo alcune persone non hanno bisogno di presentazioni, e il BDFL ("il benevolo dittatore a vita") è una di queste. Dropbox è entusiasta di dare il benvenuto a Guido, il creatore del linguaggio di programmazione Python, e nostro amico di lunga data.
Sono passati cinque anni da quando il nostro primo prototipo è stato salvato come dropbox.py, e Guido e la community di Python sono stati cruciali nell'aiutarci a risolvere sfide che hanno riguardato più di cento milioni di persone.
Dunque accogliamo Guido qui a Dropbox con ammirazione e gratitudine. Guido ha ispirato tutti noi, ha svolto un ruolo fondamentale nel modo in cui Dropbox unisce i prodotti, i dispositivi e i servizi nella vostra vita. Siamo felici di averlo nel nostro team.

Nello stesso annuncio, Drew rende merito a Python, il suo linguaggio di programmazione preferito, elogiandolo per la sua portabilità, semplicità, flessibilità ed eleganza:

Fin dall'inizio fu chiaro che Dropbox doveva supportare tutti i più importanti sistemi operativi. Storicamente, questo aspetto rappresentava una sfida importante per gli sviluppatori: ogni piattaforma richiedeva diversi tool di sviluppo, diversi linguaggi di programmazione, e gli sviluppatori si trovavano nella condizione di dover scrivere lo stesso codice numerose volte.
Non avevamo il tempo di farlo, e fortunatamente Python venne in nostro soccorso. Diversi anni prima, Python era diventato il mio linguaggio di programmazione preferito perché presentava un equilibrio tra semplicità, flessibilità ed eleganza. Queste qualità di Python, e il lavoro della community nel supportare ogni importante piattaforma, ci permise di scrivere il codice solo una volta, per poi eseguirlo ovunque. Python e la community hanno influenzato la più importante filosofia che sta dietro lo sviluppo di Dropbox: realizzare un prodotto semplice che riunisca la vostra vita.

Chi conosce Python non può certo dare torto a Drew Houston. È universalmente noto che programmare con Python è un piacere, per via della sua sintassi chiara e leggibile, che lo rende semplice e facile da imparare, ma anche perché già con i *tipi built-in* e con la libreria standard si può fare quasi tutto, e solamente per esigenze molto specialistiche ci si affida a librerie di terze parti. A tutto ciò dobbiamo aggiungere un altro importante pregio: Python è un linguaggio multiparadigma, ovvero permette di utilizzare diversi paradigmi di programmazione: metaprogrammazione, procedurale, funzionale, a oggetti e scripting.

NOTA

Per onestà intellettuale, mi sento in dovere di informarvi su una importante controindicazione nell'uso di Python: crea dipendenza. Una volta che avrete imparato a programmare con Python, vi sarà veramente difficile poterne farne a meno.

Detto ciò, Drew è stato lungimirante nelle sue scelte e, dopo meno di un anno dall'assunzione di Guido e dall'annuncio del raggiungimento dei 100 milioni di utenti, Dropbox nel novembre del 2013 ha raddoppiato, arrivando a quota 200 milioni.
Concludiamo questa sezione con un suggerimento: la lettura del *Python Advocacy HOWTO*, che si può consultare alla seguente pagina sul sito ufficiale: http://docs.python.org/3/howto/advocacy.html.

La comunità italiana di Python

L'Italia gioca un ruolo di primo piano per quanto riguarda lo sviluppo di Python. Dal 2011 sino al 2013 la Conferenza Europea di Python (EuroPython, www.europython.eu) si è tenuta a Firenze ed è stata organizzata in modo impeccabile dell'Associazione di Promozione Sociale Python Italia.

La comunità italiana è numerosa e, come quella internazionale, è molto amichevole ed entusiasta di condividere le proprie esperienze, fornire aiuto e organizzare assieme eventi e incontri.
Far parte della comunità è semplicissimo: basta iscriversi alla mailing list, seguendo le istruzioni disponibili alla pagina http://www.python.it/comunita/mailing-list/.

Implementazioni di Python

C'è un'importante precisazione che dobbiamo fare in merito al termine *Python*. Questo, infatti, viene utilizzato per indicare due cose strettamente correlate, ma comunque distinte: il *linguaggio Python* e l'*interprete Python*.
Il linguaggio Python, come si può intuire, è l'analogo di una lingua come può essere l'italiano o l'inglese, composto quindi da un insieme di parole, da regole di sintassi e da una semantica. Il codice ottenuto dalla combinazione di questi elementi si dice che è scritto nel linguaggio di programmazione Python. Questo codice, di per sé, non ha alcuna utilità. Diviene utile nel momento in cui si ha uno strumento che lo analizza, lo capisce e lo esegue. Questo strumento è l'interprete Python.
Quindi, quando installiamo Python o usiamo il programma `python`, stiamo installando o usando l'interprete, ovvero il tool che ci consente di eseguire il codice scritto nel linguaggio di programmazione Python. L'interprete Python è, a sua volta, scritto in un linguaggio di programmazione. In realtà vi sono diversi interpreti Python, ciascuno dei quali è implementato in modo differente rispetto agli altri:

- *CPython*: l'interprete classico, implementato in C (www.python.org);
- *PyPy*: interprete in RPython (Restricted Python) e compilatore Just-in-Time (www.pypy.org);
- *IronPython*: implementato su piattaforma .NET (www.ironpython.net);
- *Jython*: implementato su piattaforma Java (www.jython.org);
- *Stackless Python*: ramo di CPython che supporta i microthreads (www.stackless.com).

L'implementazione classica, quella che troviamo già installata nelle distribuzioni Linux e nei computer Mac, e presente sul sito ufficiale, è la CPython. Questa è l'implementazione di riferimento e viene comunemente chiamata *Python*. Per questo motivo nel prosieguo del libro, se non specificato diversamente, quando si parlerà dell'interprete Python, della sua installazione e del suo avvio, e quando si faranno considerazioni sull'implementazione, si intenderà sempre CPython.

Modalità di esecuzione del codice Python

Come detto, la teoria che studieremo in questo libro è aggiornata a Python 3.4, per cui il consiglio è di provare gli esempi del libro utilizzando questa versione. In ogni caso,

teniamo a mente che il codice che funziona con la versione 3.x funziona anche con ogni versione 3.y, con y maggiore di x.
L'interprete Python sui sistemi Unix-like si trova usualmente in */usr/bin/python* o */usr/local/bin/python*:

```
$ which python
/usr/local/bin/python
```

mentre sulle macchine Windows tipicamente si trova in *C:\Python*.

NOTA

Nel corso del libro faremo ampio utilizzo di comandi Unix e li accompagneremo con delle note per spiegarne il significato. Inoltre l'intera Appendice A è dedicata ai comandi Unix utilizzati nel libro. Abbiamo appena visto il comando `which`. Questo prende come argomento il nome di un file eseguibile che si trova nei percorsi di ricerca e ne restituisce il percorso completo. In altri termini, ci dice dove si trova il programma:

```
$ which python
/usr/bin/python
$ which skype
/usr/bin/skype
```

Possiamo avviare l'interprete dando il comando `python` da terminale:

```
$ python
```

L'interprete può essere avviato con varie opzioni, sulle quali possiamo documentarci eseguendo Python con lo switch `-h`:

```
$ python -h
```

Come vedremo, l'interprete si comporta allo stesso modo di una shell Unix: quando viene chiamato con lo standard input connesso a un terminale, legge ed esegue i comandi in modo interattivo, mentre quando viene chiamato passandogli il nome di un file come argomento, oppure gli viene reindirizzato lo standard input da file, legge ed esegue i comandi contenuti nel file. Inoltre, quando viene utilizzato lo switch `-c`, esegue delle istruzioni passategli sotto forma di stringa. Vediamo in dettaglio tutte e tre queste modalità di esecuzione.

Modalità interattiva

Quando i comandi vengono letti da un terminale, si dice che l'interprete è in esecuzione in *modalità interattiva*. Per utilizzare questa modalità si esegue da linea di comando `python` senza argomenti:

```
$ python
Python 3.4.0a2 (default, Sep 10 2013, 20:16:48)
[GCC 4.7.2] on linux
Type “help”, “copyright”, “credits” or “license” for more information.
>>>
```

L'interprete stampa un messaggio di benvenuto, che inizia con il numero della versione di Python in esecuzione, per poi mostrare il prompt principale, usualmente indicato con tre segni di maggiore. Quando una istruzione o un blocco di istruzioni continua su più linee, viene stampato il prompt secondario, indicato con tre punti:

```
>>> for i in range(10):
...     print(i, end=' ')
...
0 1 2 3 4 5 6 7 8 9
```

Sia dal prompt principale sia da quello secondario si può cancellare l'input e ritornare al prompt principale digitando il carattere di interruzione, tipicamente **Control-C** o **DEL**:

```
>>> for i in range(10 # Digito `Control-C`...
KeyboardInterrupt
>>>
```

Si può uscire dalla modalità interattiva (con stato di uscita 0) dando il comando `quit()` nel prompt principale, oppure digitando un carattere di *EOF* (**Control-D** sui sistemi Unix-like, **Control-Z** su Windows).
Poiché la modalità interattiva è molto utile sia per provare in modo veloce del codice, sia per effettuare dell'introspezione sugli oggetti, è preferibile utilizzare un ambiente più confortevole della semplice modalità interattiva built-in. Vi sono varie opzioni, tra le quali IPython, bpython, DreamPie o l'ambiente di sviluppo IDLE (Integrated Development Environment), compreso nelle distribuzioni Python standard. Queste supportano funzioni avanzate, come la tab completion con introspezione sugli oggetti, la colorazione della sintassi, l'esecuzione dei comandi di shell e la history dei comandi.

NOTA

Nei sistemi Unix-like la libreria *GNU readline* consente di avere la history e la tab completion anche nella modalità interattiva built-in. Per avere la tab completion, dobbiamo importare i moduli `rlcompleter` e `readline` e chiamare `readline.parse_and_bind('tab: complete')`:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
```

NOTA

A questo punto (con un doppio TAB) abbiamo la tab completion:

```
>>> import builtins
>>> builtins.a
builtins.abs(    builtins.all(    builtins.any(    builtins.ascii(
>>> builtins.a
```

Indubbiamente, dover eseguire le tre istruzioni sopra riportate ogni volta che si avvia la modalità interattiva è una scocciatura. Conviene, quindi, automatizzare questa procedura creando un file contenente le tre istruzioni e poi assegnare questo file alla variabile d'ambiente `PYTHONSTARTUP`. In questo modo, ogniqualvolta avviamo la modalità interattiva, Python come prima cosa eseguirà le istruzioni contenute nel file. Vediamo come fare quanto appena illustrato in un sistema Unix-like con shell bash. Creiamo sulla nostra home un file di startup, chiamato, ad esempio *.pyrc*, contenente le tre istruzioni Python:

```
import rlcompleter
import readline
readline.parse_and_bind('tab: complete')
```

Aggiungiamo la seguente linea sul file *.bashrc* presente nella home (se il file non esiste lo creiamo):

```
export PYTHONSTARTUP=$HOME/.pyrc
```

Ecco fatto. Non dobbiamo fare altro che aprire un terminale (oppure dare il comando `source ~/.pyrc` sul terminale già aperto) e avviare l'interprete interattivo.

I file contenenti codice Python sono detti *moduli Python*. Parleremo dei moduli nella sezione *I moduli come contenitori di istruzioni* di questo capitolo, e in modo approfondito nella sezione *Organizzazione, installazione e distribuzione delle applicazioni* del Capitolo 4. Come vedremo, i moduli assolvono a vari compiti. Ad esempio, un modulo può essere utilizzato come contenitore logico nel quale definire classi e funzioni, oppure come script, cioè con lo scopo di essere eseguito al fine di ottenere un risultato o di compiere alcune azioni.
Un programma Python è composto da uno o più moduli, e quindi la sua architettura può spaziare da un semplice script fino a un complesso programma strutturato in centinaia di moduli. I moduli Python tipicamente hanno suffisso *.py*, e sia su Windows che sui sistemi Unix-like possono essere eseguiti dal prompt dei comandi, passando il nome del file come argomento. Ad esempio, il file *myfile.py*:

```
$ cat myfile.py
print('www.python.org')
```

può essere eseguito da un terminale Unix, come mostrato di seguito:

```
$ python myfile.py
www.python.org
```

NOTA

Il comando `cat` dei sistemi Unix mostra il contenuto di uno o più file di testo (o dello standard input) su un terminale testuale e produce sullo standard output la concatenazione del loro contenuto.

Poiché su Windows i file *.py* sono associati direttamente all'eseguibile *python.exe*, è possibile eseguirli semplicemente facendo un doppio clic su di essi. Sui sistemi Unix-like, invece, come per gli script di shell, i moduli possono essere direttamente eseguiti inserendo lo *shebang* nella prima linea del file:

```
$ cat myfile.py
#! /usr/bin/env python
print('www.python.org')
```

e rendendoli poi eseguibili:

```
$ chmod +x myfile.py
$ ./myfile.py
www.python.org
```

NOTA

Il comando `chmod` dei sistemi Unix modifica i permessi di file e directory. Ad esempio, se vogliamo che il proprietario del file abbia i permessi di esecuzione, passiamo `+x` come argomento di `chmod`:

```
$ chmod +x myfile.py
```

A questo punto il file può essere eseguito direttamente da linea di comando. Visto che come prima linea del file è presente lo shebang `#! /usr/bin/env python`, il file verrà eseguito con Python:

```
$ /home/marco/temp/myfile.py
www.python.org
```

Per maggiori informazioni sul comando `chmod` e sulla notazione `./` utilizzata nel precedente esempio, possiamo consultare l'Appendice A.

Per default in Python 3 l'interprete utilizza la codifica UTF-8 per decodificare il contenuto dei moduli. Se, però, vogliamo scrivere il codice utilizzando una codifica differente, siamo liberi di farlo, a patto di informare di ciò l'interprete. Possiamo specificare la codifica da noi utilizzata inserendo nel modulo un commento speciale al di sotto dello shebang, contenente `coding:nome` o `coding=nome`:

```
$ cat myfile.py
#! /usr/bin/env python
# -*- coding: ascii -*-
print('www.python.org')
```

NOTA

Tipicamente questo commento speciale, come nell'esempio appena mostrato, contiene anche i tre caratteri `-*-` sia prima che dopo l'indicazione della codifica. Questa sintassi è ispirata alla notazione che si utilizza con Emacs per definire le variabili locali a un file. Per Python, però, questi simboli non hanno alcun significato, poiché il programma va a verificare che nel commento sia presente l'indicazione `coding:nome` o `coding=nome`, e non bada al resto.

Esecuzione di stringhe passate da linea di comando

Quando l'interprete viene chiamato utilizzando lo switch `-c`, allora accetta come argomento una stringa contenente istruzioni Python, e le esegue:

```
$ python -c "import sys; print(sys.platform)"
linux
```

Utilizzeremo spesso questa modalità di esecuzione nel resto del libro.

Introduzione al linguaggio

In questa sezione parleremo di alcuni aspetti caratteristici di Python, dei principali tipi built-in, della definizione di funzioni e classi, dei file, della libreria standard e dei moduli. Questa vuole essere solo una breve introduzione al linguaggio, poiché discuteremo di questi argomenti diffusamente e in modo approfondito nel resto del libro.

Indentazione del codice

In Python un blocco di codice nidificato non è delimitato da parole chiave o da parentesi graffe, bensì dal simbolo di due punti e dall'indentazione stessa del codice:

```
>>> for i in range(2): # Alla prima iterazione `i` varrà 0 e alla seconda 1
...     if i % 2 == 0:
...         print("Sono all'interno del blocco if, per cui...")
```

```
...         print(i, "è un numero pari.\n")
...         continue # Riprendi dalla prima istruzione del ciclo `for`
...     print("Il blocco if è stato saltato, per cui...")
...     print(i, "è un numero dispari\n")
...
Sono all'interno del blocco if, per cui...
0 è un numero pari.

Il blocco if è stato saltato, per cui...
1 è un numero dispari
```

NOTA

Per blocco di codice nidificato intendiamo il codice interno a una classe, a una funzione, a una istruzione `if`, a un ciclo `for`, a un ciclo `while` e così via. I blocchi di codice nidificati, e solo loro, sono preceduti da una istruzione che termina con il simbolo dei due punti. Vedremo più avanti che i blocchi nidificati sono la suite delle istruzioni composte o delle relative clausole.

L'indentazione deve essere la stessa per tutto il blocco, per cui il numero di caratteri di spaziatura (spazi o tabulazioni) è significativo:

```
>>> for i in range(2):
...     print(i) # Indentazione con 4 spazi
...    print(i + i) # Indentazione con 3 spazi...
  File "<stdin>", line 3
    print(i + i) # Indentazione con 3 spazi...
                                              ^
IndentationError: unindent does not match any outer indentation level
```

Nella PEP-0008 si consiglia di utilizzare quattro spazi per ogni livello di indentazione, e di non mischiare mai spazi e tabulazioni.

NOTA

Se come editor usiamo Vim e vogliamo che i blocchi di codice vengano automaticamente indentati con quattro spazi, allora inseriamo nel file di configurazione *.vimrc* le seguenti linee:

```
filetype plugin indent on
set ai ts=4 sts=4 et sw=4
```

Spesso capiterà, inoltre, di dover editare file scritti da altri, contenenti indentazioni con tabulazioni. In questi casi, con Vim, una volta effettuata la configurazione appena illustrata, possiamo convertire tutte le tabulazioni del file in spazi mediante il comando `retab`.

Utilizzare insieme spazi e tabulazioni per indentare istruzioni nello stesso blocco è un errore:

```
>>> for i in range(2):
...     print(i) # Indentazione con spazi
...     print(i) # Indentazione con TAB
  File "<stdin>", line 3
    print(i) # Indentazione con TAB
                                ^
TabError: inconsistent use of tabs and spaces in indentation
```

NOTA

In Python 3, a differenza di Python 2, è un errore anche passare da spazi a tabulazioni (o viceversa) quando si cambia livello di indentazione, come mostrato nell'Appendice B, nella sezione intitolata *Uso inconsistente di spazi e tabulazioni*.

L'indentazione, quindi, è un requisito del linguaggio e non una questione di stile, e questo implica che tutti i programmi Python abbiano lo stesso aspetto. Ma forse un giorno le cose cambieranno, e potremmo utilizzare anche le parentesi graffe per delimitare i blocchi di codice:

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
```

No, non c'è proprio possibilità, per cui le discussioni sul come indentare e posizionare i delimitatori dei blocchi non hanno motivo di esistere (vedi PEP-0666), e tutto ciò è perfettamente in linea con lo Zen di Python, secondo il quale, come vedremo tra poco, "dovrebbe esserci un modo ovvio, e preferibilmente uno solo, di fare le cose". Questo unico modo ovvio di fare le cose è detto *pythonic way*.
Infine, come ormai sappiamo, non è necessario terminare le istruzioni con un punto e virgola, ma è sufficiente andare su una nuova linea. Il punto e virgola è invece necessario se si vogliono inserire più istruzioni su una stessa linea:

```
>>> a = 'Per favore, non farlo mai!'; print(a)
Per favore, non farlo mai!
```

Inserire più istruzioni sulla stessa linea è un pessimo stile di programmazione, e infatti è sconsigliato anche dalla PEP-0008. Nel resto del libro useremo il punto e virgola solamente per eseguire delle stringhe da linea di comando:

```
$ python -c "a = 'python'; print(a)"
python
```

NOTA

Esistono vari tool che consentono di verificare che il nostro codice rispetti la PEP-0008. Possiamo fare un veloce check online con http://pep8online.com/, oppure, in alternativa, possiamo usare *pep8* (https://pypi.python.org/pypi/pep8) o *pyflakes*. Sono disponibili anche vari plugin che consentono di effettuare il check con Vim.

Gli oggetti built-in

Quando un programma viene eseguito, Python, a partire dal codice, genera delle strutture dati, chiamate *oggetti*, sulle quali basa poi tutto il processo di elaborazione. Gli oggetti vengono tenuti nella memoria del computer, in modo da poter essere richiamati quando il programma fa riferimento a essi. Nel momento in cui non servono più, un particolare meccanismo, chiamato *garbage collector*, provvede a liberare la memoria da essi occupata.
Questa prima descrizione di un oggetto è probabilmente troppo astratta per farci percepire di cosa realmente si tratti, ma non preoccupiamoci, è sufficiente per gli scopi di questa sezione e ne vedremo di più formali e concrete al momento opportuno.
Gli oggetti che costituiscono il cuore di Python, detti *oggetti built-in*, vengono comunemente distinti nelle seguenti categorie: *core data type*, *funzioni built-in*, *classi* e tipi di *eccezioni built-in*. Qui faremo solo una breve panoramica sugli oggetti built-in, poiché a essi dedicheremo l'intero Capitolo 2.

Core data type

Ciò che chiamiamo *core data type* è semplicemente l'insieme dei principali *tipi* built-in. Questi possono essere raggruppati in quattro categorie:

- *numeri*: interi (tipo `int`), booleani (`bool`), complessi (`complex`), floating point (`float`);
- *insiemi*: rappresentati dal tipo `set`;
- *sequenze*: stringhe (`str` e `byte`), liste (`list`) e tuple (`tuple`);
- *dizionari*: rappresentati dal tipo `dict`.

I tipi del core data type appartengono a una categoria di oggetti chiamati *classi*, o anche *tipi*. La caratteristica dei tipi, come suggerisce la parola, è quella di rappresentare un tipo di dato generico, dal quale poter creare oggetti specifici di quel tipo, chiamati *istanze*. Ad esempio, dal tipo `str` possiamo creare le istanze `"python"`, `"Guido"` e `"abc"`; dal tipo `int` le istanze `22` e `77`; dal tipo `list` le istanze `[1, 2, 3]` e `['a', 'b', 'c', 'd']`, e via dicendo per gli altri tipi.
Quindi diremo che una stringa di testo è un oggetto di tipo `str`, o, equivalentemente, che è una istanza del tipo `str`. Allo stesso modo, diremo che un intero è un oggetto

di tipo `int`, o, in modo equivalente, che è una istanza del tipo `int`, e via dicendo per i floating point, le liste ecc.
Alcuni oggetti sono semplici da immaginare poiché di essi abbiamo chiaro in mente sia il concetto di valore che quello di tipo, come nel caso delle istanze dei tipi numerici:

```
>>> 22 + 33
55
>>> 1.5 * 1.5
2.25
>>> (1 + 1j) / (1 - 1j)
1j
```

Il loro tipo, come quello di qualsiasi altro oggetto, si può ottenere dalla classe built-in `type`:

```
>>> type(22) # L'oggetto rappresentativo del numero `22` è una istanza del tipo `int`
<class 'int'>
>>> type(1.5) # L'oggetto rappresentativo del numero `1.5` è una istanza del tipo `float`
<class 'float'>
>>> type(1 + 1j) # L'oggetto rappresentativo del numero `1 + 1j` è una istanza del tipo `complex`
<class 'complex'>
```

Gli oggetti sono sempre caratterizzati da un tipo, mentre solo ad alcuni di essi possiamo associare in modo intuitivo un valore. Oltre al tipo, altri elementi caratteristici degli oggetti sono l'*identità* e gli *attributi*.
L'identità è rappresentata da un numero che li identifica in modo unico, e viene restituita dalla funzione built-in `id()`:

```
>>> id(22)
136597616
>>> id(1.5)
3074934816
>>> id(1j)
3073986144
```

Gli attributi sono degli identificativi accessibili per mezzo del delimitatore *punto*:

```
>>> a = 11
>>> b = 11.0
>>> c = 11 + 0j
>>> a.bit_length() # Restituisce il numero minimo di bit necessario per rappresentare l'intero
4
>>> b.as_integer_ratio() # Rappresentazione di `b` come rapporto di interi
(11, 1)
>>> c.imag # Parte immaginaria di un numero complesso
0.0
```

In questo esempio `bit_length`, `as_integer_ratio` e `imag` sono attributi, rispettivamente, di `a`, `b` e `c`. Gli attributi sono strettamente legati al tipo di un oggetto. Ad esempio, tutti

gli oggetti di tipo `str` hanno l'attributo `str.upper()` che restituisce una versione maiuscola della stringa:

```
>>> s1 = 'ciao'
>>> s2 = 'hello'
>>> s1.upper()
'CIAO'
>>> s2.upper()
'HELLO'
```

e tutti gli oggetti di tipo `list` hanno l'attributo `list.sort()` che riordina gli elementi della lista:

```
>>> mylist1 = [3, 1, 2]
>>> mylist1.sort()
>>> mylist1
[1, 2, 3]
>>> mylist2 = ['c', 'a', 'b']
>>> mylist2.sort()
>>> mylist2
['a', 'b', 'c']
```

Se un identificativo può essere seguito dalle parentesi tonde `()`, si dice che l'oggetto al quale fa riferimento è *chiamabile* (*callable*). Quando applichiamo le parentesi tonde all'identificativo, diciamo che stiamo *chiamando* l'oggetto. Consideriamo, ad esempio, un numero complesso:

```
>>> c = 1 + 2j
>>> type(c)
<type 'complex'>
```

Tutti i numeri complessi hanno il seguente attributo:

```
>>> c.conjugate
<built-in method conjugate of complex object at 0x7f85e5f8e130>
```

e questo è chiamabile. Quando lo chiamiamo, restituisce il complesso coniugato del numero:

```
>>> c.conjugate()
(1-2j)
```

Possiamo scoprire se un oggetto è chiamabile grazie alla funzione built-in `callable()`:

```
>>> callable(c.conjugate)
True
```

Se proviamo a chiamare un oggetto che non è chiamabile, otteniamo un errore:

```
>>> c.real # Parte reale del numero complesso
1.0
>>> callable(c.real)
False
>>> c.real() # Non è chiamabile...
Traceback (most recent call last):
    ...
TypeError: 'float' object is not callable
```

Gli oggetti chiamabili si differenziano da quelli non chiamabili per il fatto che consentono di eseguire una serie di operazioni, ovvero un blocco di istruzioni. Le funzioni, ad esempio, sono degli oggetti chiamabili. Per chiarire meglio il concetto, consideriamo la funzione built-in `sum`:

```
>>> sum
<built-in function sum>
```

Questa è un oggetto chiamabile:

```
>>> callable(sum)
True
```

e se la chiamiamo esegue la somma degli elementi dell'oggetto che le passiamo come argomento:

```
>>> sum([1, 2, 3]) # Restituisce la somma 1 + 2 + 3
6
```

Se in una chiamata non dobbiamo passare degli argomenti, dobbiamo ugualmente utilizzare le parentesi:

```
>>> s = 'ciao'
>>> s.upper # Non stiamo effettuando la chiamata
<built-in method upper of str object at 0x7fe031fd8a08>
>>> s.upper() # Stiamo effettuando la chiamata
'CIAO'
```

Le parentesi, infatti, indicano che vogliamo eseguire le operazioni che competono all'oggetto chiamabile.
Gli attributi chiamabili sono detti *metodi*. In base a quanto abbiamo appena detto, la differenza tra i metodi e gli altri attributi è che i primi possono essere chiamati per eseguire delle operazioni, mentre i secondi no. Consideriamo ancora il numero complesso `c = 1 + 2j`:

```
>>> c = 1 + 2j
```

I suoi attributi `c.real` e `c.imag` non sono dei metodi e quindi non possono essere chiamati. L'attributo `c.conjugate` è, invece, un metodo e quando viene chiamato esegue l'operazione `c.real - c.imag` e poi restituisce il risultato:

```
>>> c.real # Attributo non chiamabile, che rappresenta la parte reale del numero complesso
1.0
>>> c.imag # Attributo non chiamabile, che rappresenta la parte immaginaria del numero complesso
2.0
>>> c.conjugate # Attributo chiamabile (metodo)
<built-in method conjugate of complex object at 0x7f85e5f8e130>
>>> c.conjugate() # Chiamata al metodo: calcola il complesso coniugato e restituisce il valore
(1-2j)
```

NOTA

Nel corso del libro, quando nel testo scriveremo l'identificativo di un metodo o di una funzione, useremo le parentesi tonde. Ad esempio, scriveremo `c.conjugate()` e `id()` e non `c.conjugate` e `id` per indicare il metodo `coniugate()` dei numeri complessi e la funzione built-in `id()`. Quando, invece, parleremo delle classi, nonostante siano oggetti chiamabili, non useremo le parentesi tonde, per cui scriveremo, ad esempio, `type` e non `type()`. Nel Capitolo 6, quando parleremo del modello a oggetti di Python e delle metaclassi, capiremo perché ha senso fare la distinzione tra oggetti che sono classi e oggetti che non lo sono.

In sostanza, i metodi sono delle funzioni e infatti vengono definiti come tali, come vedremo tra breve nella sezione *Definire le classi*.
Se questi concetti vi sembrano troppo astratti, non preoccupatevi, li riprenderemo varie volte nel corso del libro e li affronteremo in modo esaustivo nel Capitolo 5, quando introdurremo la programmazione orientata agli oggetti e vedremo nel dettaglio i vari tipi di metodo.
La funzione built-in `dir()` restituisce una lista dei nomi degli attributi più significativi di un oggetto:

```
>>> dir(33)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__
delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__',
'__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__',
'__gt__', '__hash__', '__index__', '__init__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__
or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__
subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

Tutti gli attributi che iniziano e finiscono con un doppio underscore vengono chiamati *attributi speciali* o anche *attributi magici*. Vedremo il significato di qualcuno di essi in questo capitolo e nei prossimi due, mentre dei restanti parleremo in modo approfondito nel Capitolo 6.
La funzione built-in `hasattr()` ci dice se un oggetto ha un certo attributo:

```
>>> hasattr(33, 'as_integer_ratio') # Gli interi non hanno l'attributo 'as_integer_ratiò
False
>>> hasattr(33.0, 'as_integer_ratio') # I float hanno l'attributo 'as_integer_ratiò
True
```

Vediamo ora i tipi del core data type. Come abbiamo già detto, questa sarà solo una breve introduzione, poiché li tratteremo in dettaglio nel Capitolo 2.

Le stringhe di testo

Le stringhe di testo in Python sono rappresentate da una sequenza di caratteri Unicode di lunghezza arbitraria, racchiusi tra apici singoli, virgolette, tripli apici singoli o triple virgolette:

```
>>> s1 = 'stringa di testo' # Non può essere spezzato su più linee
>>> s2 = "stringa di testo" # Non può essere spezzato su più linee
>>> s3 = '''stringa
... di testo''' # Può essere spezzato su più linee
>>> s4 = """stringa
... di testo""" # Può essere spezzato su più linee
```

Una stringa di testo è un oggetto di tipo `str`:

```
>>> s = "esempio di stringa di testo"
>>> type(s)
<class 'str'>
```

e i suoi elementi sono *ordinati*, nel senso che a ciascuno di essi è associato un numero intero chiamato *indice*, che vale 0 per l'elemento più a sinistra e aumenta progressivamente di una unità per i restanti, andando in modo ordinato da sinistra verso destra.
Per questo motivo, le stringhe appartengono alla categoria delle *sequenze*, la quale comprende tutti i tipi built-in che rappresentano dei *contenitori ordinati* di lunghezza arbitraria (stringhe, liste e tuple).
Il metodo `str.index()` restituisce l'indice della prima occorrenza dell'elemento passatole come argomento:

```
>>> s.index('i')
5
```

È possibile anche compiere l'operazione inversa, ovvero ottenere un elemento dalla stringa utilizzando come chiave di ricerca l'indice.
Questa operazione è detta *indicizzazione* (*indexing*), e viene compiuta utilizzando la seguente sintassi:

```
>>> s[5]
'i'
```

Un'altra operazione che possiamo compiere con gli indici è lo *slicing*. Questa consente di ottenere gli elementi di una stringa compresi tra due indici arbitrari:

```
>>> s[2:6] # Elementi di `s` compresi tra gli indici 2 e 6 (escluso)
'empi'
```

Le operazioni di indicizzazione e di slicing sono comuni a tutti gli oggetti appartenenti alla categoria delle sequenze. Questa categoria rientra all'interno di un'altra ancora più generica, quella degli *oggetti iterabili*, dei quali parleremo nella sezione *Oggetti iterabili, iteratori e contesto di iterazione* al termine di questo capitolo. Gli oggetti iterabili supportano una operazione detta di *spacchettamento* (*unpacking*), che consiste nell'assegnare gli elementi dell'oggetto iterabile a delle etichette, nel seguente modo:

```
>>> a, b, c, d = s[2:6] # Equivale a 4 istruzioni: a='e'; b='m'; c='p'; d='i'
>>> a
'e'
>>> c
'p'
```

Vediamo qualche altra operazione che possiamo compiere con le stringhe:

```
>>> s.startswith('esem') # La stringa `s` inizia con `esem`?
True
>>> s.count('di') # Conta il numero di occorrenze della stringa 'di' all'interno di `s`
2
>>> s.title() # Restituisce una copia di `s` avente le iniziali delle parole maiuscole
'Esempio Di Stringa Di Testo'
>>> s.upper()
'ESEMPIO DI STRINGA DI TESTO'
>>> '  ciao  '.strip() # Rimuove gli spazi all'inizio e alla fine della stringa
'ciao'
>>> 'python' * 3 # Ripetizione di stringhe
'pythonpythonpython'
>>> '44' + '44' # Concatenazione di stringhe
'4444'
>>> int('44') + int('44') # Converto le stringhe in interi, poi li sommo
88
```

Gli oggetti di tipo `str` non possono essere modificati, e per questo motivo si dice che sono degli oggetti *immutabili*:

```
>>> s[0] = 'E'
Traceback (most recent call last):
    ...
TypeError: 'str' object does not support item assignment
```

Le liste

Anche le *liste* appartengono alla categoria delle sequenze, ma, a differenza delle stringhe, possono contenere oggetti di qualunque tipo. Vengono rappresentate inserendo gli elementi tra parentesi quadre, separati con una virgola:

```
>>> mylist = ['ciao', 100, 33] # Lista contenente gli elementi 'ciao', 100 e 33
>>> mylist.index(100)
1
>>> mylist[1] # Indicizzazione
100
>>> mylist[0:2] # Slicing
['ciao', 100]
>>> 'ciao' in mylist # L'elemento 'ciao' è contenuto in `mylist`?
True
>>> mylist.reverse() # Modifica la lista invertendo l'ordine dei suoi elementi
>>> mylist
[33, 100, 'ciao']
>>> mylist.pop() # Restituisce e poi rimuove l'ultimo elemento della lista
'ciao'
>>> mylist
[33, 100]
>>> mylist.append(5) # Inserisce un elemento nella lista
>>> mylist
[33, 100, 5]
>>> mylist.sort() # Modifica la lista riordinando i suoi elementi
>>> mylist
[5, 33, 100]
>>> mylist.extend(['a', 'b', 'c']) # Estende la lista con un'altra lista
>>> mylist
[5, 33, 100, 'a', 'b', 'c']
>>> mylist + ['python'] # Operazione di concatenazione
[5, 33, 100, 'a', 'b', 'c', 'python']
>>> mylist * 3 # Operazione di ripetizione
[5, 33, 100, 'a', 'b', 'c', 5, 33, 100, 'a', 'b', 'c', 5, 33, 100, 'a', 'b', 'c']
>>> mylist[2] = 77 # Le liste sono oggetti mutabili
>>> mylist
[5, 33, 77, 'a', 'b', 'c']
>>> a, b, c = [1, 2, 3] # Spacchettamento
>>> b
2
```

Come ormai sappiamo, le liste possono essere modificate. Per questo motivo si dice che sono *oggetti mutabili*.

Le tuple

Altri tipi di oggetti analoghi alle liste sono le *tuple*. Anch'esse appartengono alla categoria delle sequenze e possono contenere oggetti di qualunque tipo, ma, a differenza delle liste, sono immutabili, e vengono rappresentate inserendo gli elementi tra parentesi tonde piuttosto che tra parentesi quadre:

```
>>> t = (1, 'due', [1, 'lista'], 'due', ())
>>> t.count('due') # Restituisce il numero di occorrenze dell'elemento 'due'
2
>>> t[1]
'due'
>>> t[1] = 'uno' # Le tuple sono oggetti immutabili
Traceback (most recent call last):
    ...
TypeError: 'tuple' object does not support item assignment
>>> a, b = (1, -2) # La tupla viene spacchettata per l'assegnamento
>>> b
-2
>>> a, b = 1, -2 # Viene creata la tupla (1, -2) e poi spacchettata
>>> a, type(b) # Elementi separati da virgola: restituisce una tupla
(1, <class 'int'>)
```

I dizionari

I *dizionari* sono dei contenitori aventi per elementi delle coppie *chiave:valore*. Vengono rappresentati inserendo gli elementi tra parentesi graffe, separandoli con una virgola:

```
>>> d = {'cane': 'bau', 'gatto': 'miao', 'uno': 1, 2: 'due'}
```

A differenza degli oggetti appartenenti alla categoria delle sequenze, i dizionari non sono dei contenitori ordinati, per cui la ricerca viene effettuata per chiave e non per indice:

```
>>> d['cane'] # Ricerca per chiave
'bau'
>>> 2 in d # La chiave `2` è nel dizionario?
True
>>> 'bau' in d # La chiave 'bau' è nel dizionario?
False
>>> d.pop('cane') # Restituisce il valore e elimina l'elemento
'bau'
>>> d # La coppia `'cane': 'bau'` è stata eliminata
{2: 'due', 'gatto': 'miao', 'uno': 1}
>>> d['uno'] += 1 # Viene incrementato di `1` il valore corrispondente alla chiave 'uno'
>>> d
{2: 'due', 'gatto': 'miao', 'uno': 2}
```

I set

I set sono dei contenitori non ordinati di oggetti unici e immutabili, e hanno le stesse proprietà degli insiemi matematici. I set vengono rappresentati inserendo gli elementi tra parentesi graffe, separandoli con una virgola:

```
>>> s1 = {1, 'uno', 'one', 'numero'}
{1, 'uno', 'numero', 'one'}
>>> s2 = {1, 'uno', 'two'}
>>> s1 | s2 # Unione dei due insiemi
{1, 'uno', 'two', 'numero', 'one'}
>>> s1 & s2 # Intersezione dei due insiemi
{1, 'uno'}
>>> s1 - s2 # Differenza: elementi di `s1` che non appartengono a `s2`
{'numero', 'one'}
```

I set sono oggetti mutabili:

```
>>> s2.add(2) # I set sono mutabili
>>> s2
{1, 2, 'two', 'uno'}
```

Gli elementi di un set devono, però, essere oggetti immutabili:

```
>>> s = {1, 2, []} # Non possono contenere oggetti mutabili, come una lista
Traceback (most recent call last):
    ...
TypeError: unhashable type: 'list'
```

Funzioni e classi built-in

Abbiamo già visto qualche classe e funzione built-in, come `type` e `id()`. Vediamone delle altre all'opera, in modo da apprezzare già da subito la potenza degli oggetti built-in di Python:

```
>>> any([1, 0, 3]), any((0, 0, 0)) # Vi sono elementi non nulli nella sequenza?
(True, False)
>>> all((1, 1, 1)), all([1, 1, 0]) # Gli elementi della sequenza sono tutti non nulli?
(True, False)
>>> bin(15), hex(15), oct(15) # Rappresentazione binaria, esadecimale e ottale
('0b1111', '0xf', '0o17')
>>> len(['a', 'b', 3, 'd']) # Restituisce la lunghezza della sequenza
4
>>> max([1, 2, 5, 3]), min([1, 2, 5, 3]) # Valori massimo e minimo di una sequenza
(5, 1)
>>> abs(1 - 1j) # Valore assoluto di un numero
1.4142135623730951
>>> sum([1, 2, 3, 4]) # Somma gli elementi della sequenza
10
>>> str(['a', 'b', 'c', 1, 2]) # Converte un oggetto in stringa
"['a', 'b', 'c', 1, 2]"
```

```
>>> eval('2 * 2') # Valuta una espressione contenuta in una stringa
4
>>> exec('a = 2 * 2') # Esegue istruzioni contenute in una stringa
>>> a
4
```

Un primo sguardo alla libreria standard e al concetto di modulo

Probabilmente abbiamo sentito dire che Python ha le batterie incluse (*batteries included*). Questa frase viene utilizzata dai pythonisti per dire che, una volta installato Python, si ha già tutto ciò che serve per essere produttivi, poiché, oltre ad avere degli oggetti built-in di altissimo livello, si ha a disposizione anche una libreria standard (*standard library*) che copre praticamente tutto.
La libreria standard di Python è strutturata in moduli, ognuno dei quali ha un preciso ambito di utilizzo. Ad esempio, il modulo `math` ci consente di lavorare con la matematica, il modulo `datetime` con le date e con il tempo, e via dicendo. Per importare un modulo si utilizza la parola chiave `import`, dopodiché, come per tutti gli altri oggetti, possiamo accedere ai suoi attributi tramite il delimitatore punto:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(0), math.log(1)
(0.0, 0.0)
```

Possiamo utilizzare `import` in combinazione con la parola chiave `from` per importare dal modulo solo gli attributi che ci interessano, evitando così di scrivere ogni volta il nome del modulo:

```
>>> from math import pi, sin, log
>>> pi
3.141592653589793
>>> sin(0), log(1)
(0.0, 0.0)
```

Altri moduli della libreria standard, dei quali faremo ampio utilizzo nel corso del libro, sono `datetime`, `sys` e `os`. Il modulo `datetime`, come abbiamo detto, ci consente di lavorare con le date e con il tempo:

```
>>> from datetime import datetime
>>> d = datetime.now() # È un oggetto che rappresenta il tempo attuale
>>> d.year, d.month, d.day # Attributi per l'anno, il mese e il giorno
(2012, 12, 4)
>>> d.minute, d.second # Minuti e secondi
(31, 29)
>>> (datetime.now() - d).seconds # Numero di secondi trascorsi da quando ho creato `d`
16
```

Il modulo sys si occupa degli oggetti legati all'interprete e all'architettura della nostra macchina:

```
>>> import sys
>>> sys.platform
'linux'
>>> sys.version # Versione di Python
'3.4.0a2 (default, Sep 10 2013, 20:16:48) \n[GCC 4.7.2]'
```

Il modulo os, del quale parleremo nella sezione *Esercizio conclusivo* al termine di questo capitolo, fornisce il supporto per interagire con il sistema operativo:

```
>>> import os
>>> os.environ['USERNAME'], os.environ['SHELL'] # L'attributo `os.environ` è un dizionario
('marco', '/bin/bash')
>>> u = os.uname()
>>> u.sysname, u.version
('Linux', '#67-Ubuntu SMP Thu Sep 6 18:18:02 UTC 2012')
```

Gli ambiti di utilizzo della libreria standard non si fermano certamente qui. Vi sono dei moduli che forniscono il supporto per l'accesso a Internet e il controllo del web browser:

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://www.python.org/community/diversity/'):
...     text = line.decode('utf-8')
...     if '<title>' in text:
...         print(text)
...
  <title>Diversity</title>

>>> import webbrowser
>>> url = 'http://www.python.org/dev/peps/pep-0001/'
>>> webbrowser.open(url) # Apre un url usando il browser di default
True
```

Altri moduli forniscono una interfaccia per i *file wildcard*:

```
>>> import glob
>>> glob.glob('*')
['_templates', 'ch1', 'prefazione.rst', 'index.rst', '_static', 'conf.py']
>>> glob.glob('*.py')
['conf.py']
```

e tanti altri ancora.
Questa è stata solo una brevissima introduzione alla libreria standard, di cui parleremo ancora nel resto del libro. Se siamo impazienti, possiamo dare subito uno sguardo alla documentazione online, andando alla pagina web http://docs.python.org/3/library/index.html.