

- 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**
DigitalLifeStyle

Phyton

Guida completa

Marco Buttu



Phyton | Guida completa

Autore: Marco Buttu

Collana: PROGRAMMARECON

Publisher: Fabrizio Comolli

Editor: Marco Aleotti

Progetto grafico: Roberta Venturieri

Immagine di copertina: © Hlubokidzianis | Dreamstime.com

ISBN: 978-88-6895-009-5

Copyright © 2014 **LSWR Srl**

Via Spadolini, 7 - 20141 Milano (MI) - www.lswr.it

Prima edizione digitale: maggio 2014

Nessuna parte del presente libro può essere riprodotta, memorizzata in un sistema che ne permetta l'elaborazione, né trasmessa in qualsivoglia forma e con qualsivoglia mezzo elettronico o meccanico, né può essere riprodotta o registrata altrimenti, senza previo consenso scritto dell'editore, tranne nel caso di brevi citazioni contenute in articoli di critica o recensioni.

La presente pubblicazione contiene le opinioni dell'autore e ha lo scopo di fornire informazioni precise e accurate. L'elaborazione dei testi, anche se curata con scrupolosa attenzione, non può comportare specifiche responsabilità in capo all'autore e/o all'editore per eventuali errori o inesattezze.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive aziende. L'autore detiene i diritti per tutte le fotografie, i testi e le illustrazioni che compongono questo libro, salvo quando diversamente indicato.

Sommario

INTRODUZIONE

1. I FONDAMENTI DEL LINGUAGGIO

- Introduzione a Python
- Introduzione al linguaggio
- Gli elementi del codice Python
- Architettura di un programma Python
- La Python Virtual Machine
- Etichette e oggetti
- Tipologie di errori
- Oggetti iterabili, iteratori e contesto di iterazione
- Esercizio conclusivo

2. IL CUORE DEL LINGUAGGIO

- Numeri
- Operazioni e funzioni built-in utilizzabili con gli oggetti iterabili
- Gli insiemi matematici
- Dizionari
- Le sequenze
- Esercizio conclusivo

3. FUNZIONI, GENERATORI E FILE

- Definizione e chiamata di una funzione
- Funzioni anonime
- Introspezione sulle funzioni
- Generatori
- File
- Esercizio conclusivo

4. MODULI, PACKAGE, AMBIENTI VIRTUALI E APPLICAZIONI

- Moduli
- Namespace, scope e risoluzione dei nomi
- Installazione dei package
- Ambienti virtuali
- Esercizio conclusivo

5. CLASSI E PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

- Classi e istanze
- Un primo sguardo all'overloading
- La composizione

- L'ereditarietà
- I decorator
- I metodi e le property
- Introduzione ai design pattern
- Le eccezioni
- L'istruzione with e i context manager
- Esercizio conclusivo

6. ATTRIBUTI MAGICI, METACLASSI E TEST DRIVEN DEVELOPMENT

- Il modello a oggetti di Python
- Gli attributi magici
- Metaclassi
- Test driven development
- Esempio pratico di utilizzo del test driven development
- Le enumerazioni
- Esercizio conclusivo

APPENDICE A - DESCRIZIONE DEI COMANDI UNIX-LIKE UTILIZZATI NEL LIBRO

- cat
- chmod
- cut
- diff
- echo
- find
- grep
- head
- ln
- ls
- mkdir
- more
- mv
- pwd
- rm
- sed
- source
- tail
- tar
- time
- touch
- tree
- wc
- wget
- which
- zip
- I metacaratteri
- Variabili d'ambiente

APPENDICE B - PRINCIPALI PUNTI DI ROTTURA TRA PYTHON 2 E

PYTHON 3

Incompatibilità tra le due versioni

Porting automatico da Python 2 a Python 3

APPENDICE C - IL BUFFERING DEI FILE

Introduzione

Python è un linguaggio di programmazione multiplatforma, 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 data-type, 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.

I fondamenti del linguaggio

In questo capitolo costruiremo solide **fondamenta** pratico-teoriche che consentiranno di essere **subito produttivi**. Introduciamo 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 **programmatore 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 multiplatforma, 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 multiplatforma 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 dall'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/mailling-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, ogniquale volta 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 `conjugate()` 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_', '_getattr_', '_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_ratio'
False
>>> hasattr(33.0, 'as_integer_ratio') # I float hanno l'attributo 'as_integer_ratio'
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 passato 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)
'emp'
```

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>.

Definire le funzioni

Le *funzioni* vengono definite mediante l'istruzione `def`:

```
>>> def square(x):
...     return x * x
...
>>> square(2)
4
```

Se non è presente l'istruzione `return`, la funzione restituisce implicitamente l'oggetto `None`:

```
>>> def foo():
...     pass
...
>>> a = foo()
>>> type(a)
<class 'NoneType'>
```

Parleremo delle funzioni in modo esaustivo e dettagliato nel Capitolo 3.

Definire le classi

I tipi del core data type sono degli oggetti incredibilmente utili, che ci consentono di essere immediatamente produttivi e di scrivere buona parte del

codice senza sentire il bisogno di avere a disposizione oggetti di altro tipo. Ma ognuno di noi ha le proprie esigenze, e vorrebbe utilizzare degli oggetti che gli facilitino il più possibile il lavoro.

Supponiamo, ad esempio, di voler scrivere un'applicazione che richieda all'utente di compiere un'azione di registrazione inserendo i suoi dati, come nome e cognome. In questo caso sarebbe comodo avere degli oggetti con le caratteristiche di una persona, che abbiano, ad esempio, come attributi proprio un nome, un cognome e un metodo che restituisca il tempo trascorso dal momento della registrazione. Per fare ciò il linguaggio mette a disposizione l'istruzione `class`. Questa consente di creare un oggetto chiamato *classe* o *tipo*, tramite il quale possiamo creare altri oggetti detti *istanze* della classe. Se facciamo mente locale, ricorderemo che abbiamo introdotto questi concetti nella sezione *Core data type*, dedicata proprio alle istanze dei tipi del core data type.

Tutte le istanze di una classe hanno i medesimi attributi, per cui, ad esempio, possiamo creare una classe che definisce una generica persona, e utilizzare le sue istanze per rappresentare gli utenti che si registrano nella nostra applicazione:

```
>>> from datetime import datetime # Supporto per date e tempo
>>> class Person:
...     def __init__(self, name, surname):
...         self.name = name # Attributo che rappresenta il nome
...         self.surname = surname # Attributo che rappresenta il cognome
...         self.registration_date = datetime.now() # Data di registrazione
...     def since(self): # L'istruzione `def` nel corpo di una classe definisce un metodo
...         return (datetime.now() - self.registration_date).seconds
...
>>> p1 = Person('Tim', 'Peters')
>>> p2 = Person('Raymond', 'Hettinger')
>>> p1.name, p1.surname
('Tim', 'Peters')
>>> p2.since() # Restituisce il numero di secondi trascorsi da quando `p2` si è registrato
15
>>> p1.since() # Restituisce il numero di secondi trascorsi da quando `p1` si è registrato
23
```

Le istanze di una classe `MyClass` sono anche dette *oggetti di tipo* `MyClass`, per cui diremo che le istanze `p1` e `p2` del tipo `Person` sono oggetti di tipo `Person`.

Il metodo `__init__()` inizia e finisce con due underscore, per cui, come abbiamo già detto nella sezione *Core data type*, è un metodo speciale, infatti viene chiamato automaticamente da Python subito dopo la creazione dell'istanza con

lo scopo di inicializzarla. Il meccanismo è il seguente: quando Python incontra l'istruzione `p1 = Person('Tim', 'Peters')`, prima crea l'istanza, poi fa la chiamata al metodo `__init__()` passandogli automaticamente l'istanza appena creata come primo argomento e, infine, restituisce l'istanza assegnandola a `p1`. Quindi, in sostanza, Python ci evita di fare manualmente la chiamata `Person.__init__(p1, 'Tim', 'Peters')`.

Il primo parametro di `__init__()`, che viene usualmente chiamato `self`, è presente anche nella definizione del metodo `Person.since()`. Infatti la chiamata a questo metodo tramite una istanza, come nel caso di `p2.since()`, è equivalente alla chiamata `Person.since(p2)`, quindi, anche in questo caso, l'istanza viene automaticamente passata al metodo come primo argomento e assegnata a `self`. La funzione built-in `isinstance()` ci dice se un oggetto è una istanza di una certa classe:

```
>>> isinstance(p1, Person), isinstance(p2, Person)
(True, True)
```

Se non avete delle basi di programmazione orientata agli oggetti e non vi è chiaro quanto detto sul concetto di classe e sul suo ambito di utilizzo, non preoccupatevi: il [Capitolo 5](#) sarà interamente dedicato a questi argomenti.

Un primo sguardo ai file

I file in Python vengono gestiti ad alto livello con la funzione built-in `open()`. Questa consente di creare un cosiddetto *file object*, ovvero un oggetto tramite il quale possiamo interagire con il file che risiede nel file-system, che chiameremo *file sottostante*. Gli attributi del file object ci forniscono informazioni come il nome del file e la codifica che si sta utilizzando per leggerlo:

```
>>> f = open('/etc/hostname')
>>> f.name, f.encoding
('/etc/hostname', 'UTF-8')
```

Essi ci consentono, inoltre, di interagire con il file sottostante, ad esempio per leggerne il contenuto:

```
>>> f.read() # Restituisce tutto il contenuto del file sotto forma di stringa
'my-laptop'
```

Possiamo ottenere il contenuto del file anche come lista di linee, mediante il

metodo `readline()`:

```
>>> f = open('/etc/networks')
>>> lines = f.readlines() # Restituisce una lista contenente le linee del file
>>> lines
['# symbolic names for networks, see networks(5) for more information\n', 'link-local 169.254.0.0\n']
>>> lines[1]
'link-local 169.254.0.0\n'
```

Come possiamo osservare, le linee sono stringhe che terminano con `\n`. Questa rappresentazione non corrisponde alla sequenza dei due caratteri `\` e `n`, ma è un carattere singolo:

```
>>> len('\n') # La lunghezza è pari a 1 perché corrisponde a un singolo carattere
1
```

Questo fa parte di un insieme di caratteri, detti *caratteri di controllo*, ai quali non è associato alcun simbolo grafico. Ne parleremo in modo approfondito nel prossimo capitolo. Per il momento ci interessa solo sapere che `\n` è utilizzato per rappresentare una interruzione di linea, chiamata in inglese *newline*. Quindi, quando leggiamo il contenuto di un file, ogni interruzione di linea viene rappresentata con `\n`, e ogni `\n` all'interno di una stringa viene visto come una interruzione di linea quando questa viene scritta su file o mostrata a video con `print()`:

```
>>> print('python\n3\nabcd')
python
3
abcd
```

Possiamo eseguire delle operazioni sulle linee di un file in modo semplicissimo. Infatti, quando in un ciclo `for` si itera su di un file, si accede automaticamente alle sue linee, una per volta:

```
>>> for line in open('/etc/legal'):
...     print(line, end=")
...
```

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in `/usr/share/doc/*/copyright`.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

I file hanno delle modalità di apertura, come ad esempio quelle in lettura e scrittura. Per specificare la modalità si passa a `open()` una stringa come secondo argomento. La stringa "w" indica scrittura (*writing mode*), mentre la stringa "r" indica lettura (*reading mode*). Quando la modalità non è specificata, per default il file viene aperto in lettura:

```
>>> f = open('myfile', 'w')
>>> f.write('scriviamo qualcosa nel file...') # Restituisce il numero di caratteri scritti
30
>>> f = open('myfile') # Equivalente a `f = open('myfile', 'r')`
>>> f.read()
'scriviamo qualcosa nel file...'
```

Concludiamo qui la nostra introduzione ai file. Ne parleremo in modo approfondito nel [Capitolo 3](#).

La funzione built-in `print()`

La forma generale della funzione built-in `print()` è la seguente:

```
print(obj_1, ..., obj_N, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Se non le si passa alcun oggetto, stampa una linea vuota:

```
>>> print() # Se non le si passa alcun argomento stampa una linea vuota
>>>
```

Se, invece, le si passano uno o più oggetti `obj_i`, li stampa separandoli con la stringa `sep`, che per default è uno spazio, e concludendo la stampa con la stringa `end`, che per default è un newline:

```
>>> print('a', 'b', 'c', 1, 2, 3)
a b c 1 2 3
```

Sostanzialmente la funzione `print()` non fa altro che creare la stringa:

```
str(obj_1) + sep + ... + sep + str(obj_N) + end
```

per poi passarla al metodo `write()` dell'oggetto assegnato al parametro `file`. Quindi, poiché per default si ha `file=sys.stdout`, la seguente `print()`:

```
>>> print([1, 2], 3, 'via!', sep='-')
[1, 2]-3-via!
```

è equivalente a questa chiamata al metodo `sys.stdout.write()`:

```
>>> import sys
>>> sys.stdout.write(str([1, 2]) + '-' + str(3) + '-' + 'via!' + '\n')
[1, 2]-3-via!
```

Allo stesso modo, se come file non usiamo lo standard output ma un file fisico:

```
>>> print('una lista:', [1, 2], file=open('myfile', 'w'))
>>> open('myfile').read()
'una lista: [1, 2]\n'
```

il codice equivalente alla `print()` è il seguente:

```
>>> file = open('myfile', 'w')
>>> file.write(str('una lista: ') + ' ' + str([1, 2]) + '\n')
```

Il parametro `flush` serve per forzare lo svuotamento del buffer subito dopo la stampa. Per default si ha `flush=False`, il che significa che il contenuto del buffer non viene immediatamente scritto sul file:

```
>>> f = open('myfile', 'w')
>>> print('una lista: ', [1, 2], file=f)
>>> print('seconda linea', file=f)
>>> open('myfile').read() # Il file è ancora vuoto
''
>>> f.close() # Quando il file viene chiuso il contenuto del buffer viene scritto sul file
>>> open('myfile').read() # Ora infatti possiamo leggere il contenuto del file
'una lista: [1, 2]\nseconda linea\n'
```

Se, invece, impostiamo `flush=True`, allora il buffer viene svuotato a ogni stampa:

```
>>> f = open('myfile', 'w')
>>> print('linea da scrivere sul file...', file=f, flush=True)
>>> open('myfile').read()
'linea da scrivere sul file...\n'
```

Parleremo del buffering nella sezione *I file* del Capitolo 3 e nell'Appendice C. Se volessimo effettuare un elevato numero di stampe su un file piuttosto che a video, sarebbe comodo poter evitare di passare ogni volta il file alla `print()` come argomento. Possiamo fare ciò cambiando temporaneamente `sys.stdout`, visto che questo è assegnato per default al parametro `file`:

```
>>> import sys
```

```
>>> sys.stdout = open('myfile.log', 'w') # Assegno un file allo standard output
>>> print('scrivo su file di log') # La print() scrive su myfile.log
>>> print('ancora su file di log...') # Scrivo ancora su `myfile.log`
>>> sys.stdout.close() # Svuoto il buffer, il contenuto viene scritto su `myfile.log`
>>> open('myfile.log').read() # Il contenuto è il testo scritto con le print()
'scrivo su file di log\nancora su file di log...\n'
>>> sys.stdout = sys.__stdout__ # Ripristino lo standard output
>>> print('eccoci qua') # Adesso la `print()` stampa nuovamente a video
eccoci qua
```

Ottenere informazioni sugli oggetti

Il termine *introspezione* viene utilizzato per riferirsi alla capacità di ottenere informazioni sugli oggetti. Python ha tantissimi strumenti con i quali poter fare introspezione. Molti di questi li abbiamo già incontrati, come la classe built-in `type` e la funzione built-in `id()`:

```
>>> type([1, 2, 3, 4]) # Se le si passa un oggetto, ne restituisce il tipo
<class 'list'>
>>> id('python') # La funzione built-in `id()` restituisce l'identità di un oggetto
3073967616
```

le funzioni built-in `hasattr()` e `dir()`:

```
>>> hasattr(1 + 2j, 'real') # Ci dice se un oggetto ha un certo attributo
True
>>> dir(1 + 2j) # Lista contenente i nomi degli attributi di un oggetto
['__abs__', '__add__', '__bool__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
 '__floordiv__', '__format__', '__ge__', '__getattr__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__int__',
 '__le__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__pos__', '__pow__', '__radd__',
 '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__', '__rpow__',
 '__rsub__', '__rtruediv__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', 'conjugate',
 'imag', 'real']
```

le funzioni built-in `isinstance()` e `callable()`:

```
>>> isinstance(22, int) # Ci dice se un oggetto è istanza di una certa classe
True
>>> callable(isinstance) # Ci dice se un oggetto è chiamabile
True
```

e la parola chiave `in`:

```
>>> 'yth' in 'python'
True
>>> 1 in ['a', 'c', 1, 2]
True
```

Tanti altri li incontreremo più avanti, come ad esempio la parole chiave `is`, che

possiamo utilizzare per sapere se due oggetti sono identici:

```
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2, 3]
>>> l1 == l2 # Operatore di uguaglianza: verifica se due oggetti hanno lo stesso valore
True
>>> l1 is l2 # Verifica se `id(l1) == id(l2)`
False
>>> id(l1), id(l2)
(3071820748, 3072442604)
>>> l3 = l2
>>> l2 is l3
True
>>> id(l2), id(l3)
(3072442604, 3072442604)
```

Approfondiremo il concetto di identità tra breve. Per il momento concentriamoci su due importanti strumenti per l'introspezione, da utilizzare principalmente in modalità interattiva: le funzioni built-in `dir()` e `help()`.

La funzione built-in `dir()`

La funzione built-in `dir()`, come già detto, 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__', '__getattr__', '__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']
```

Lo scopo della funzione built-in `dir()` non è quello di verificare se un oggetto abbia un certo attributo, poiché non tutti gli attributi dell'oggetto sono elencati nella lista da essa restituita:

```
>>> '__name__' in dir(dict)
False
>>> dict.__name__ # Ma `dict` ha l'attributo `__name__`
'dict'
```

Per verificare se un oggetto ha un dato attributo dobbiamo usare `hasattr()`:

```
>>> hasattr(dict, '__name__')
True
```

Il principale ambito di utilizzo di `dir()` è l'ausilio al programmatore durante la fase di scrittura del codice. Quando si programma, spesso non si ricordano a memoria tutti gli attributi di un oggetto, così solitamente si consulta della documentazione con lo scopo di trovare un elenco degli attributi, sperando che dal nome si riesca a dedurre il significato.

Ad esempio, stiamo scrivendo del codice e a un certo punto abbiamo bisogno di ottenere da una stringa la corrispondente versione maiuscola. Sappiamo che le stringhe hanno un metodo che restituisce una copia della stringa in versione maiuscola, ma non ne ricordiamo il nome. A questo punto `dir()` è la nostra migliore amica. Avviamo l'interprete in modalità interattiva e controlliamo l'elenco degli attributi delle stringhe:

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
 '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Trovato! È l'attributo `str.upper()`:

```
>>> 'python'.upper()
'PYTHON'
```

Quindi dobbiamo utilizzare la funzione built-in `dir()` come se fosse l'indice analitico della nostra documentazione. Analogamente a un indice analitico, `dir()` ci fornisce un elenco dei nomi, ma non ci dà informazioni sul loro significato.

La funzione built-in `help()` e le stringhe di documentazione

Per poter ottenere delle informazioni su un attributo possiamo utilizzare la funzione built-in `help()`. Per capire quando e come utilizzarla, consideriamo un esempio pratico. Supponiamo di dover scrivere del codice per poter leggere le linee del seguente file:

```
30;44;99;88
11;17;16;50
33;91;77;15
```

in modo da ottenere per ciascuna linea la somma dei suoi elementi (i numeri separati dal simbolo di punto e virgola). Sappiamo, dalla sezione *Un primo*

sguardo ai file, che possiamo leggere le linee del file una alla volta, sotto forma di stringhe di testo. Vogliamo capire se le stringhe hanno un metodo che consente di spezzarle in corrispondenza di un dato carattere (il punto e virgola, nel nostro caso). Apriamo, quindi, la shell interattiva e digitiamo `dir(str)`. Vediamo che le stringhe hanno il metodo `str.split()` che sembra fare al caso nostro, per cui facciamo qualche tentativo per capire se questo si comporta come vogliamo:

```
>>> '33;91;77;15'.split()
['33;91;77;15']
>>> '33 91 77 15'.split()
['33', '91', '77', '15']
```

Abbiamo capito che `str.split()` spezza la stringa in corrispondenza degli spazi. Ma noi vorremmo spezzare la stringa in corrispondenza del punto e virgola, per cui ci interessa sapere se possiamo passare a `str.split()` un argomento che indichi il carattere di separazione. A questo punto entra in gioco la funzione `help()`. Se chiamiamo `help(str.split)` sulla shell interattiva, viene stampata a video la documentazione del metodo `str.split()`:

```
split(...)
S.split(sep=None, maxsplit=-1) -> list of strings
Return a list of the words in S, using sep as the
delimiter string. If maxsplit is given, at most maxsplit
splits are done. If sep is not specified or is None, any
whitespace string is a separator and empty strings are
removed from the result.
(END)
```

La documentazione ci dice che `str.split()` accetta un primo argomento opzionale che indica l'elemento in corrispondenza del quale spezzare la stringa, infatti:

```
>>> '33;91;77;15'.split(';')
['33', '91', '77', '15']
```

Adesso dobbiamo eseguire la somma degli elementi. Abbiamo visto, nella sezione *Funzioni e classi built-in*, che la funzione built-in `sum()` effettua la somma degli elementi di una sequenza. Per poterli sommare dobbiamo prima convertirli in numeri interi:

```
>>> mylist = []
>>> for item in '33;91;77;15'.split(';'):
...     mylist.append(int(item))
...
```

```
>>> mylist
[33, 91, 77, 15]
```

Ecco una sintassi chiamata *list comprehension*, che ci consente di fare questo in modo più conciso:

```
>>> mylist = [int(item) for item in '33;91;77;15'.split(';')]
>>> mylist
[33, 91, 77, 15]
```

Possiamo utilizzare questa sintassi per generare in linea l'argomento da passare a `sum()`:

```
>>> sum([int(item) for item in '33;91;77;15'.split(';')])
216
```

In realtà in questo caso possiamo utilizzare una sintassi ancora più semplice, detta *generator expression*, che vedremo nell'esercizio conclusivo di questo capitolo e in modo dettagliato nel Capitolo 3:

```
sum(int(item) for item in '33;91;77;15'.split(';'))
216
```

Adesso abbiamo tutte le informazioni che servono per scrivere il nostro programma:

```
$ cat myscript.py
for line in open('data.txt'):
    result = sum(int(data) for data in line.split(';'))
    print(result)
```

```
$ cat data.txt
30;44;99;88
11;17;16;50
33;91;77;15
$ python myscript.py
261
94
216
```

La funzione built-in `help()` ricava la maggior parte delle informazioni andando a leggere delle particolari stringhe di documentazione, dette *documentation string*, o, in modo più semplice, *docstring*. Le docstring vengono utilizzate per documentare i moduli, le classi, i metodi e le funzioni, e vanno posizionate in cima al corpo di questi. La PEP-0257 suggerisce di utilizzare

sempre le triple virgolette per racchiudere il testo delle docstring, anche quando queste sono contenute in una sola linea. Consideriamo la seguente funzione:

```
>>> def doubling(obj):  
...     """Raddoppia l'oggetto."""  
...     return obj * 2
```

La prima riga nel corpo di `doubling()` è una docstring, e viene assegnata all'attributo `__doc__`:

```
>>> doubling.__doc__  
"Raddoppia l'oggetto."
```

La funzione built-in `help()` ottiene le informazioni sull'oggetto andando a leggere questo attributo, come mostra il risultato di `help(doubling)`:

```
Help on function doubling in module __main__:  
doubling(obj)  
Raddoppia l'oggetto.
```

NOTA

Si osservi che abbiamo chiamato `help(doubling)` e non `help(doubling())`. Quando passiamo a `help()` un oggetto chiamabile non dobbiamo usare le parentesi, altrimenti l'oggetto viene chiamato e, anziché passare a `help()` la funzione, le passiamo l'oggetto da essa restituito.

Possiamo ottenere la documentazione dei metodi dei tipi del core data type qualificandoli con la classe o anche con l'istanza:

```
>>> s = 'python'  
>>> print(s.upper.__doc__)  
S.upper() -> str
```

```
Return a copy of S converted to uppercase.  
>>> print(str.upper.__doc__)  
S.upper() -> str
```

```
Return a copy of S converted to uppercase.
```

Lo Zen di Python

Il 3 giugno del 1999, Patrick Phalen mandò un messaggio alla python-list, dal titolo *The Python Way*. Sostanzialmente suggeriva a Guido van Rossum e Tim Peters di scrivere un documento con 10-20 aforismi che potessero riassumere lo spirito del linguaggio, una sorta di Zen di Python:

Sarebbero disposti Guido e Tim Peters a collaborare alla stesura di un piccolo documento -- chiamiamolo "The Python Way" in mancanza di un titolo migliore -- che esponga 10-20 linee guida da offrire a coloro che si avvicinano a Python da altri linguaggi, e che vogliono immediatamente trovare un modo per usarlo propriamente nelle situazioni più insidiose (implementare closure ecc.)?

Ciò che ho in mente è una specie di introduzione molto breve, una sorta di "Elementi di Stile" di Strunk & White per Python, che elenchi le tipiche raccomandazioni fondamentali per scrivere il codice in accordo con lo spirito del linguaggio. Sto parlando di una sorta di Zen di Python -- qualcosa da citare e contemplare quando le lamentele del tipo "fix Python now" diventano un po' troppe.

Messaggio originale:

<http://mail.python.org/pipermail/python-list/1999-June/014096.html>.

Il giorno dopo Tim Peters rispose a Patrick elencando 19 aforismi e concluse dicendo:

Eccoci: esattamente 20 tesi pythoniche, contando quella che lascio scrivere a Guido. Se le risposte a ogni problema di design con Python non risultano ovvie dopo averle lette, bene, allora mi ritiro.

Messaggio originale:

<http://mail.python.org/pipermail/python-list/1999-June/001951.html>.

Il ventesimo non è mai arrivato, e questi 19 aforismi sono stati raccolti nella PEP-0020, *The Zen of Python*, scritto nel 2004 dallo stesso Tim Peters.

NOTA

Un altro importante documento, che indica delle linee guida di comportamento, è il *Code of Conduct* (codice di condotta), approvato

nell'aprile del 2013 dalla PSF. A differenza dello Zen di Python, che si focalizza sul codice, il codice di condotta è centrato sul comportamento da tenersi all'interno della comunità Python, o, meglio, descrive un membro della comunità Python come: aperto (*open*), premuroso nei confronti degli altri membri (*considerate*) e rispettoso (*respectful*). Il documento è disponibile alla pagina: <http://www.python.org/psf/codeofconduct/>.

Anche il modulo `this` della libreria standard è dedicato allo Zen. In realtà questo modulo è più che un semplice richiamo ai 19 aforismi: è un *easter egg*. È scritto in modo tale da visualizzare lo Zen quando viene importato e, allo stesso tempo, rappresentare un chiaro esempio di cosa non si deve fare. L'origine del modulo `this` risale al 2001, quando, in occasione della decima International Python Conference (ancora non esisteva la Python Conference – PyCon), si stava scegliendo uno slogan da stampare sulle t-shirt. Vi erano varie proposte, ma alla fine la scelta cadde su *import this*. Scelto lo slogan, restava da implementare un modulo chiamato `this`. Questo è il risultato:

```
$ cat /usr/local/lib/python3.4/this.py
s = ""Gur Mra bs Clguba, ol Gvz Crgref
```

```
Ornhgvshy vf orggre guna htyl.
Rkcyvpvg vf orggre guna vzcyvpvg.
Fvzcyr vf orggre guna pbzcyrk.
Pbzcyrk vf orggre guna pbzcyvpngrq.
Syng vf orggre guna arfgrq.
Fcnefr vf orggre guna qrafr.
Ernqnovyvgl pbhagf.
Fcrpvny pnfrf nera'g fcrpvny rabhtu gb oernx gur ehyrf.
Nygubhtu cenpgvpnyvgl orngf chevgl.
Reebef fubhyq arire cnff fvyragyl.
Hayrff rkcyvpvgyl fvyraprq.
Va gur snpr bs nzovthvgl, ershfr gur grzcgngvba gb thrff.
Gurer fubhyq or bar-- naq cersrenoyl bayl bar --boivbhf jnl gb qb vg.
Nygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu.
Abj vf orggre guna arire.
Nygubhtu arire vf bsgra orggre guna *evtug* abj.
Vs gur vzcyrzragngvba vf uneq gb rkcyntva, vg'f n onq vqrn.
Vs gur vzcyrzragngvba vf rnfl gb rkcyntva, vg znl or n tbbq vqrn.
Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!""
```

```
d = {}
for c in (65, 97):
    for i in range(26):
        d[chr(i+c)] = chr((i+13) % 26 + c)

print(''.join([d.get(c, c) for c in s]))
```

Dubito che qualcuno capisca il significato di tutto ciò. Il modulo è stato scritto in questo modo appositamente, come esempio per far capire quanto sia importante seguire lo *Zen di Python*:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Questa è la traduzione in italiano:

Lo Zen di Python, di Tim Peters

Bello è meglio che brutto.
Esplicito è meglio che implicito.
Semplice è meglio che complesso.
Complesso è meglio che complicato.
Lineare è meglio che nidificato.
Sparso è meglio che denso.
La leggibilità conta.
I casi speciali non sono abbastanza speciali per infrangere le regole.
Anche se la praticità batte la purezza.
Gli errori non dovrebbero mai passare sotto silenzio.
A meno che non vengano esplicitamente messi a tacere.
In caso di ambiguità, rifiuta la tentazione di indovinare.
Ci dovrebbe essere un modo ovvio -- e preferibilmente uno solo -- di fare le cose.
Anche se questo modo potrebbe non essere ovvio da subito, a meno che non siate olandesi.
Ora è meglio che mai.
Sebbene mai sia spesso meglio che *proprio adesso*.
Se l'implementazione è difficile da spiegare, l'idea è pessima.
Se l'implementazione è facile da spiegare, l'idea può essere buona.
I namespace sono una grandiosa idea, usiamoli il più possibile!

Se ci siamo convinti della validità di queste regole, allora possiamo passare

alla prossima sezione; in caso contrario, sarebbe meglio se continuassimo con la lettura:

```
>>> this.__doc__ # Nessuna documentazione...
>>> dir(this)
['__builtins__', '__cached__', '__doc__', '__file__', '__initializing__', '__loader__', '__name__', '__package__', 'c', 'd', 'i', 's']
>>> this.c
97
>>> this.d
{'V': 'I', 'W': 'J', 'T': 'G', 'U': 'H', 'R': 'E', 'S': 'F', 'P': 'C', 'Q': 'D', 'Z': 'M', 'X': 'K', 'Y': 'L', 'F': 'S', 'G': 'T', 'D': 'Q', 'E': 'R', 'B': 'O', 'C': 'P', 'A': 'N', 'N': 'A', 'O': 'B', 'L': 'Y', 'M': 'Z', 'J': 'W', 'K': 'X', 'H': 'U', 'I': 'V', 'v': 'i', 'w': 'j', 't': 'g', 'u': 'h', 'r': 'e', 's': 'f', 'p': 'c', 'q': 'd', 'z': 'm', 'x': 'k', 'y': 'l', 'f': 's', 'g': 't', 'd': 'q', 'e': 'r', 'b': 'o', 'c': 'p', 'a': 'n', 'n': 'a', 'o': 'b', 'l': 'y', 'm': 'z', 'j': 'w', 'k': 'x', 'h': 'u', 'i': 'v'}
>>> this.i
25
>>> this.s
"Gur Mra bs Clguba, ol Gvz Crgref\n\nOrnhgvshy vf orggre guna htyl.\nRkcyvpvg vf orggre guna vzcyvpvg.\nFvzcyr vf orggre guna pbzcyrk.\nPbzcyrk vf orggre guna pbzcyvpngrq.\nSyng vf orggre guna arfgrq.\nFcnefr vf orggre guna grafr.\nErnqnovyvgf pbhagf.\nFcrpvny pnfrf nera'g fcrpvny rabhtu gb oernx gur ehryf.\nNygubhtu cenpgvpnyvgl orngf chevgl.\nReebef fubhyq arire cnff fvyragyl.\nHayrff rkcyvpvgyl fvyraprq.\nVa gur snpr bs nzovthvgl, ershfr gur grzcgngvba gb thrff.\nGurer fubhyq or bar-- naq cersrenoyl bayl bar --boivbhf jnl gb qb vg.\nNygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu.\nAbj vf orggre guna arire.\nNygubhtu arire vf bsgra orggre guna *evtug* abj.\nVs gur vzcyzragngvba vf uneq gb rkcyynva, vg'f n onq vqrn.\nVs gur vzcyzragngvba vf rnfl gb rkcyynva, vg znl or n tbbq vqrn.\nAnzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!"
>>> out = "".join([c in this.d and this.d[c] or c for c in this.s])
>>> print(out) # Difficile capire ciò che abbiamo fatto...
```

Python, inoltre, segue la nota filosofia di Unix, *doing one thing well*, secondo la quale ogni programma deve fare una sola cosa, ma bene. Si osservi, però, che *fare una cosa bene* non significa farla in modo perfetto. La perfezione nel mondo reale non esiste, per cui, in accordo con il motto *la praticità batte la purezza*, dobbiamo fare le cose *abbastanza bene*, come suggerisce anche Alex Martelli nel suo talk intitolato *Good enough is good enough*: <http://pyvideo.org/video/1738/good-enough-is-good-enough>.

Gli elementi del codice Python

Se guardiamo un file contenente del codice Python, vedremo che, come ogni file di testo, è composto da una serie di linee, che chiameremo *linee fisiche*. Le linee fisiche non hanno significato per l'interprete Python, ma servono semplicemente per organizzare visivamente il codice in modo leggibile. Ad esempio, il file *myfile.py* contiene due linee fisiche:

```
$ cat myfile.py
import sys
print(sys.platform)
```

Ciò che ha significato per l'interprete sono le *linee logiche*. Una linea logica tipicamente corrisponde a una linea fisica, ma, volendo, è possibile suddividere una linea fisica in più linee logiche, separando queste con un carattere di punto e virgola. Ad esempio, il seguente file *myfile.py* contiene tre linee fisiche e sei linee logiche:

```
$ cat myfile.py
a = 33 + 44; b = a + 50; print(b) # Linea fisica composta da 3 linee logiche
import math # Linea fisica composta da una linea logica
a, b = math.sin(0), math.cos(0); print(a, b) # Linea fisica composta da due linee logiche
```

Probabilmente capiterà raramente di vedere più linee logiche nella stessa linea fisica, poiché è un pessimo stile di programmazione, giustamente sconsigliato dalla PEP-0008.

È anche possibile ripartire una linea logica su più linee fisiche. Questo lo si vede spesso, e lo si può fare sia utilizzando il carattere di backslash, come nell'esempio seguente:

```
>>> a = 10 + 20 + \
... 30 + 40 + \
... 50
>>> a
150
```

sia utilizzando le parentesi tonde:

```
>>> a = (10 + 20 +
... 30 + 40 +
```

```
... 50)
>>> a
150
```

e anche spezzando su più linee fisiche una lista, una tupla, un set o un dizionario:

```
>>> {1: 'uno', 2: 'due', 3: 'tre',
... 4: 'quattro', 5: 'cinque', 6: 'sei'}
{1: 'uno', 2: 'due', 3: 'tre', 4: 'quattro', 5: 'cinque', 6: 'sei'}
```

Il testo contenuto all'interno di una linea logica appartiene a una di queste categorie: *commenti*, *letterali*, *operatori*, *parole chiave*, *etichette* e *delimitatori*. Consideriamo, ad esempio, il seguente file:

```
$ cat analisi_lessicale.py
x = input('Inserisci del testo che rappresenti un numero intero: ')
num = int(x) # Converto la stringa inserita dall'utente in un numero intero
result = 10 / num if num else None
print(result)
```

Questo è strutturato in quattro linee logiche. Nella prima linea vi sono due etichette, `x` e `input`, un letterale di tipo stringa, che inizia e finisce con gli apici, e tre delimitatori (le due parentesi tonde e il simbolo di uguale). Nella seconda linea vi sono tre etichette (`num`, `int` e `x`), tre delimitatori (le due parentesi tonde e il simbolo di uguale) e un commento (il testo che inizia con un carattere di cancelletto e termina con la linea fisica). Nella terza linea vi sono due etichette (`result` e `num`), un operatore (`/`), un letterale di tipo intero (`10`), un delimitatore (il simbolo di uguale) e tre parole chiave: `if`, `else` e `None`. Infine, nella quarta linea vi sono due etichette (`print` e `result`) e due delimitatori (le parentesi tonde).

Il componente dell'interprete che si occupa di effettuare l'analisi lessicale del codice è chiamato *parser*.

I commenti

Come è noto, i commenti servono principalmente per documentare il codice al fine di facilitarne la lettura. Sono delle porzioni di testo che iniziano con il carattere di cancelletto e terminano con la fine della linea fisica. Quando l'interprete trova un commento, non esegue alcuna azione:

```
>>> a = 44 # Un altro esempio di commento
>>> for i in range(3): # Altro commento
... print(i)
```

...
0
1
2

Talvolta, però, i commenti vengono analizzati dal *parser* perché hanno un significato speciale, come nel caso del commento che indica la codifica del codice sorgente, che abbiamo visto nella sezione *Esecuzione del codice Python da file*.

Letterali

Le istanze dei tipi del core data type sono facilmente distinguibili le une dalle altre, oltre che per il *tipo*, anche per il *valore*. Questi oggetti sono il cuore del linguaggio, e rappresentano gli elementi chiave di tutti i programmi Python. Per facilitare sia la lettura che la scrittura dei programmi, si è deciso di associare a queste istanze delle rappresentazioni testuali di immediata comprensione, che consentano di determinare in modo unico sia il tipo sia il valore dell'oggetto al quale fanno riferimento. Queste rappresentazioni testuali sono dette *letterali*.

Ad esempio, un *letterale stringa* di testo è del codice delimitato da apici singoli, virgolette, tripli apici singoli o triple virgolette. Quando il programma viene eseguito, Python crea dal letterale un oggetto di tipo `str`, ovvero una struttura dati che viene caricata nella memoria del nostro computer e che rappresenta la stringa:

```
>>> s = 'letterale stringa di testo' # Testo delimitato da apici singoli
>>> type(s)
<class 'str'>
```

Una sequenza di sole cifre decimali, separate dal resto del codice da uno o più spazi, rappresenta un *letterale intero*, dal quale Python crea un oggetto di tipo `int`:

```
>>> i = 10 # Letterale di tipo intero
>>> type(i)
<class 'int'>
```

Similmente avviene per tutte le altre istanze dei tipi del core data type:

```
>>> f = 12.33 # Letterale rappresentativo di un oggetto di tipo 'float'
>>> type(f)
<class 'float'>
```

```
>>> c = 12.33 + 1j # Letterale rappresentativo di un oggetto di tipo `complex`
>>> type(c)
<class 'complex'>
>>> t = ('a', 'b', 3) # Letterale rappresentativo di un oggetto di tipo `tuple`
>>> type(t)
<class 'tuple'>
>>> mylist = ['a', 'b', 3] # Letterale rappresentativo di un oggetto di tipo `list`
>>> type(mylist)
<class 'list'>
>>> d = {1: 'uno', 2: 'due'} # Letterale rappresentativo di un dizionario
>>> type(d)
<class 'dict'>
>>> s = {'a', 'b', 'c'} # Letterale rappresentativo di un oggetto di tipo `set`
>>> type(s)
<class 'set'>
```

La definizione di letterale ci ha consentito di fornire ulteriori informazioni utili a comprendere il significato di oggetto. Gli oggetti vengono creati da Python quando il programma è in esecuzione, nel momento in cui incontra delle porzioni di codice alle quali sa che deve far corrispondere delle strutture dati in memoria, rappresentative di quel codice. Ad esempio, da una semplice porzione di codice, come il letterale `'python'`, viene creata nella memoria del computer una complessa struttura dati, che non contiene solo i caratteri della stringa, ma anche tante altre informazioni, come il tipo della struttura dati:

```
>>> s = 'python'
>>> s.__class__ # L'oggetto è una stringa
<class 'str'>
```

Grazie a queste informazioni Python capisce quali sono le azioni che può compiere su questi dati:

```
>>> s.upper()
'PYTHON'
>>> s + 33
Traceback (most recent call last):
...
TypeError: Can't convert 'int' object to str implicitly
```

Concludiamo dicendo che (ovviamente) non è possibile fare un assegnamento a un letterale:

```
>>> 22 = 44
File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> [1, 2, 3] = 44
File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Operatori

Gli operatori sono dei simboli ai quali Python associa un significato che dipende dal tipo dei suoi operandi. Ad esempio, il simbolo `+` è un operatore. Quando i suoi operandi sono dei numeri, Python effettua la loro somma, quando sono delle sequenze effettua la loro concatenazione, e quando non sa come operare segnala un errore:

```
>>> 22 + 33
55
>>> 'python' + ' ' + 'è semplicemente meraviglioso!'
'python è semplicemente meraviglioso!'
>>> [1, 2, 3] + ['a', 'b']
[1, 2, 3, 'a', 'b']
>>> [1, 2, 3] + 'ab'
Traceback (most recent call last):
...
TypeError: can only concatenate list (not "str") to list
```

Altri esempi di operatori sono i seguenti:

```
>>> a == [1, 2, 3] # Operatore di uguaglianza: `==`
True
>>> 2 ** 4 # Operatore di elevamento a potenza: `**`
16
```

Nel corso del libro spiegheremo il significato dei vari operatori.

Parole chiave

Come possiamo intuire, le parole chiave (in inglese *keywords*) sono i vocaboli del linguaggio interpretato da Python. Sono esattamente come le parole di una lingua, e servono per spiegare all'interprete ciò che vogliamo esegua. L'elenco delle parole chiave di Python è il seguente:

```
>>> help('keywords')
```

Here is a list of the Python keywords. Enter any keyword to get more help.

False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

Non è possibile cambiare il significato di una parola chiave, altrimenti Python non capirebbe più il vostro codice. Per questo motivo le parole chiave non possono comparire alla sinistra del simbolo =:

```
>>> False = 33
File "<stdin>", line 1
SyntaxError: assignment to keyword
```

Le parole chiave appartengono a due categorie, distinte a seconda che la loro prima lettera sia maiuscola o minuscola. Appartengono alla prima categoria le parole chiave `False`, `True` e `None`. Queste sono delle parole chiave speciali, poiché, quando il programma è in esecuzione, Python associa a esse un oggetto in memoria. Ad esempio, a `False` corrisponde un oggetto interpretato come il numero zero, mentre a `True` un oggetto interpretato come il numero uno:

```
>>> 10 * False, 10 + False
(0, 10)
>>> 10 * True, 10 + True
(10, 11)
>>> type(False), type(True), type(None)
(<class 'bool'>, <class 'bool'>, <class 'NoneType'>)
```

Alle restanti keyword non è associato alcun oggetto. Sono delle semplici parole che devono essere utilizzate nel codice seguendo le regole di sintassi del linguaggio, e non possiamo utilizzarle allo stesso modo del codice al quale è associato un oggetto. Non possiamo, ad esempio, ottenere il tipo della parola chiave `and`, come invece abbiamo fatto per `False`, `True` e `None`:

```
>>> type(and) # La parola chiave `and` non può essere utilizzata come una semplice etichetta
File "<stdin>", line 1
type(and)
^
SyntaxError: invalid syntax
```

Delimitatori

I delimitatori sono delle sequenze di uno, due o anche tre caratteri, che vengono utilizzate per delimitare o separare delle porzioni di codice. Ne abbiamo già incontrati diversi, come il simbolo di due punti, utilizzato per iniziare un blocco di codice:

```
>>> for i in range(5):
...     if i: # Se `i` è diverso da zero
...         print(2 / i)
```

```
...
2.0
1.0
0.6666666666666666
0.5
```

Altri delimitatori noti sono:

- il simbolo di uguale, utilizzato per l'assegnamento:

```
>>> a = 100
```

- il punto, utilizzato per accedere a un attributo:

```
>>> a.bit_length() # Minimo numero di bit necessario per rappresentare 100
7
```

- la virgola e il punto e virgola:

```
>>> a, b, c, d, e, f = 'python'
>>> a
'p'
>>> e
'o'
>>> import math; print(math.pi)
3.141592653589793
```

Vedremo i restanti delimitatori quando ne faremo uso negli esempi.

Le etichette

Tutto ciò che fa parte del codice Python e non rientra nelle precedenti categorie è una etichetta. Nel seguente codice, ad esempio, sono presenti il letterale intero 99, i delimitatori () e il testo `print`, che non è né una parola chiave, né un commento, né un segno di punteggiatura:

```
>>> print(99)
99
```

Quindi è una etichetta. La caratteristica delle etichette è che devono essere definite prima di poter essere utilizzate, perché quando il codice è in esecuzione Python fa corrispondere a esse l'oggetto a loro precedentemente assegnato. Infatti, se proviamo a utilizzare una etichetta che non è definita, Python segnala un errore:

```
>>> a + 33 # L'etichetta `a` non è definita
Traceback (most recent call last):
...
NameError: name 'a' is not defined
```

Si osservi come Python ha eseguito l'istruzione `print(99)` nonostante non avessimo definito alcuna etichetta `print`. Questa, infatti, è una etichetta già

definita dal linguaggio, alla quale è stato assegnato un oggetto di tipo `builtin_function_or_method`:

```
>>> type(print)
<class 'builtin_function_or_method'>
```

Per conoscere le etichette di tutti gli oggetti built-in possiamo controllare gli attributi del modulo `builtins`, dei quali omettiamo una parte per brevità:

```
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
...
'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'vars', 'zip']
```

La principale differenza tra le etichette, i letterali e le parole chiave è che solo le etichette possono comparire alla sinistra del simbolo di uguale:

```
>>> a = 99
>>> 'ciao' = 'python'
File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> [1, 2] = [1, 2]
File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Le etichette che si possono utilizzare in un programma Python sono quindi quelle built-in più tutti gli identificativi ai quali è stato assegnato un oggetto, sia in modo esplicito mediante una istruzione di assegnamento, sia in modo implicito.

NOTA

Si parla di assegnamenti impliciti quando a una etichetta viene assegnato un oggetto senza che si utilizzi esplicitamente il simbolo di uguale. Ad esempio, quando chiamiamo una funzione, gli argomenti vengono assegnati implicitamente ai parametri:

```
>>> def foo(a, b):
...     print(a, b)
...
>>> foo(1, 2)
1 2
```

Sono stati eseguiti implicitamente gli assegnamenti `a = 1` e `b = 2`. La stessa cosa avviene quando si importa un modulo:

```
>>> import math
>>> math
<module 'math' from '/usr/local/lib/python3.4/lib-dynload/math.cpython-34m.so'>
```

L'istruzione `import` ha provveduto a caricare in memoria il modulo e assegnarlo all'etichetta `math`.

Probabilmente abbiamo già sentito dire che una differenza fondamentale tra Python 2 e Python 3, causa di incompatibilità, è la modalità di utilizzo dell'identificativo `print`. Infatti in Python 2 `print` è una parola chiave, pertanto non può essere assegnata:

```
$ python2.7
Python 2.7.1+ (r271:86832, Sep 27 2012, 21:16:52)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print = 99 # In Python 2 `print` è una parola chiave
      File "<stdin>", line 1
print = 99
      ^
SyntaxError: invalid syntax
```

mentre in Python 3 è una semplice etichetta, per cui possiamo assegnarle un oggetto:

```
$ 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.
>>> print = ['python', 99] # In Python 3 `print` è una etichetta
>>> print
['python', 99]
```

NOTA

Per scoprire le motivazioni in base alle quali si è deciso di far diventare `print` una etichetta, possiamo consultare la PEP-3105, dal titolo *Make print a function*.

Un'altra differenza tra etichette e parole chiave di Python 2 e Python 3

riguarda `True` e `False`. Queste, infatti, sono delle parole chiave in Python 3, mentre erano delle semplici etichette in Python 2:

```
$ python2.7
Python 2.7.1+ (r271:86832, Sep 27 2012, 21:16:52)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> True = 0
>>> False = 1
>>> True * 10, False * 10
(0, 10)
```

Non possiamo utilizzare un qualsiasi identificativo per definire le etichette, ma dobbiamo seguire delle regole. Sappiamo, ad esempio, che una etichetta non può corrispondere a una parola chiave:

```
>>> from = 44
      File "<stdin>,, line 1
from = 44
^
SyntaxError: invalid syntax
>>> from_ = 44
```

Oltre a questo vincolo, le etichette non possono contenere operatori e delimitatori:

```
>>> etich-etta = 99
      File "<stdin>,, line 1
SyntaxError: can't assign to operator
>>> etich()etta = 99
      File "<stdin>,, line 1
etich()etta = 99
^
SyntaxError: invalid syntax
```

non possono iniziare con dei numeri:

```
>>> a2 = 99
>>> 2a = 99
      File "<stdin>,, line 1
2a = 99
^
SyntaxError: invalid syntax
```

e vengono valutate da Python in modo *case-sensitive*:

```
>>> aaa = 100
>>> aaa
```

100

```
>>> aAa
```

```
Traceback (most recent call last):
```

```
...
```

```
NameError: name 'aAa' is not defined
```

Per quanto riguarda le convenzioni, si può consultare la PEP-0008.

Quando una etichetta è preceduta dal delimitatore punto, si dice che è *qualificata*. Ad esempio, nell'espressione `sys.platform`, diciamo che `platform` è una *etichetta qualificata*, o anche che `platform` è qualificata tramite `sys`.

Se si vuole approfondire l'argomento dell'analisi lessicale del codice Python, si consiglia di leggere la seguente pagina sul sito ufficiale: http://docs.python.org/3/reference/lexical_analysis.html.

Architettura di un programma Python

Un programma Python è composto da *moduli*, i quali contengono al loro interno *istruzioni*, le quali processano *espressioni*. In questa sezione descriveremo questi tre elementi.

Le espressioni

Abbiamo detto che gli oggetti vengono generati dal codice in esecuzione. Proviamo a restringere il campo, individuando le porzioni di codice che danno origine alla creazione degli oggetti oppure che semplicemente fanno riferimento a essi. Queste parti di codice sono dette *espressioni*. Diamo, infatti, la seguente definizione: una *espressione* è una porzione di codice contenuta in una linea logica, che quando il programma è in esecuzione fa riferimento a un oggetto oppure lo genera.

Vediamo di capire, attraverso esempi pratici, come individuare le espressioni. Consideriamo, a tale scopo, del codice che viene correttamente eseguito, come il seguente:

```
>>> if 'p' in 'python':  
...    print('python')  
...  
python
```

Restringiamo il campo dicendo che le espressioni si trovano alla destra del simbolo di uguale e sono confinate all'interno delle linee logiche. Per scoprire se una porzione di codice è una espressione, possiamo estrapolarla dalla linea logica e assegnarla a una etichetta. Se l'assegnamento non dà luogo ad alcun errore, allora la porzione di codice è un'espressione, altrimenti non lo è.

Nel processo di estrapolazione del codice dalla linea logica, le etichette non possono essere separate dalle parentesi. Quindi `print('python')` non può essere spezzata nelle parti `print` e `('python')`, e `mylist[2]` non può essere spezzata in `mylist` e `[2]`. Iniziamo dalle stringhe `'p'` e `'python'`:

```
>>> a = 'p'  
>>> a = 'python'
```

Queste sono quindi espressioni. Vediamo la porzione di codice ottenuta dalla loro combinazione tramite la parola chiave `in`:

```
>>> a = 'p' in 'python'
```

Quindi anche 'p' in 'python' è una espressione. Vediamo la porzione di codice if 'p':

```
>>> a = (if 'p')
      File "<stdin>"," line 1
a = if 'p'
^
SyntaxError: invalid syntax
```

Abbiamo ottenuto un errore, quindi if 'p' non è una espressione. Neppure la linea logica presa nella sua interezza è una espressione:

```
>>> a = if 'p' in 'python'
      File "<stdin>"," line 1
a = if 'p' in 'python'
^
SyntaxError: invalid syntax
```

così come le restanti porzioni di codice all'interno di questa linea. Veniamo alla seconda linea:

```
>>> a = print('python') # Ok è una espressione
python
```

Anche print e ('python') sono delle espressioni, ma, come abbiamo detto, una etichetta non può essere separata dalle parentesi che la seguono, per cui print('python') è una unica espressione e non la combinazione di print e ('python'), le quali, prese singolarmente, hanno il loro significato *atomico*:

```
>>> print # Etichetta che fa riferimento a una funzione built-in
<built-in function print>
>>> ('python') # È la stringa 'python'. La tupla di un elemento è ('python',)
'python'
```

La seguente sintassi definisce una *espressione condizionale*:

```
>>> 'a' if 4 == 2 + 2 else 'b' # Una stringa non vuota è valutata True
'a'
>>> x = 'a' if 4 == 2 + 2 else 'b' # Può essere assegnata, per cui è una espressione
>>> x
'a'
```

Questa è la sua forma generale:

```
<espressione 1> if <test> else <espressione 2>
```

e restituisce espressione 1 se l'espressione test è valutata True, altrimenti restituisce espressione 2:

```
>>> result = 'expression 1' if 2 in ['a', 2, 1] else 'expression 2'
>>> result
'expression 1'
>>> result = 'expression 1' if 7 in ['a', 2, 1] else 'expression 2'
>>> result
'expression 2'
```

A differenza di altri linguaggi, in Python l'assegnamento non è una espressione:

```
>>> if a = 33: # Non possiamo usare un assegnamento come espressione di test
      File "<stdin>,, line 1
if a = 33:
^
SyntaxError: invalid syntax
>>> b = (a = 33)
      File "<stdin>,, line 1
b = (a = 33)
^
SyntaxError: invalid syntax
```

Il fatto che l'assegnamento non sia una espressione consente di evitare che in un test di confronto (ad esempio, $a = b$) una mancata battitura dia luogo a un pericoloso errore logico ($a = b$). Troviamo le motivazioni di questa e altre scelte progettuali alla pagina <http://docs.python.org/3/faq/design.html>.

Per maggiori informazioni e dettagli sulle espressioni e la loro classificazione, possiamo consultare la documentazione sul sito ufficiale: <http://docs.python.org/3/reference/expressions.html>.

Le istruzioni

In Python il codice che definisce una linea logica nella sua interezza è detto *istruzione semplice* (*simple statement*), mentre un blocco di codice composto da istruzioni semplici è detto *istruzione composta* (*compound statement*). Le istruzioni composte si caratterizzano per l'utilizzo del delimitatore di due punti, il quale indica l'inizio del blocco di codice dell'istruzione. L'unione delle istruzioni semplici e di quelle composte costituisce l'insieme delle istruzioni di Python.

Le istruzioni hanno un nome, che deriva da quello dalle parole chiave utilizzate al loro interno. Ad esempio, le istruzioni seguenti sono chiamate

istruzione semplice `import` e istruzione composta `for`:

```
>>> import math # Istruzione import
>>> for i in range(2): # Istruzione for
...     print(math.sin(i))
...     print(math.cos(i))
...
0.0
1.0
0.8414709848078965
0.5403023058681398
```

Nel corso del libro vedremo come utilizzare tutte le istruzioni, mentre in questa sezione ci limiteremo a studiare quelle che ci interessano per gli scopi di questo capitolo.

Istruzioni semplici

In base alla definizione precedente, una linea logica costituita da una espressione è una istruzione semplice. Questo, evidentemente, è l'unico caso di una istruzione che è anche una espressione. Il risultato di questa istruzione non può essere salvato; per questo motivo una espressione viene utilizzata come istruzione solo in due casi. Il primo è quando si utilizza la modalità interattiva, poiché, nonostante il risultato dell'espressione non venga salvato, esso viene scritto nello standard output e lo si può verificare visivamente:

```
>>> [1, 2, 3] + [4, 5, 6] # Linea logica costituita da una espressione
[1, 2, 3, 4, 5, 6]
```

L'altra situazione è quella delle chiamate a funzioni e metodi che non restituiscono esplicitamente un oggetto, poiché in questi casi interessa solo che vengano eseguite delle azioni e non c'è alcun risultato da salvare:

```
>>> print('Python 3') # Questa è sia istruzione che espressione
Python 3
```

Tutte le altre istruzioni semplici non sono espressioni, e pertanto non possono essere assegnate a una etichetta. Vediamone di seguito una parte.

L'assegnamento

L'*assegnamento* è una istruzione semplice che assegna l'oggetto generato dall'espressione alla destra del simbolo di `=` all'etichetta alla sinistra del

simbolo stesso:

```
>>> a = [1, 2, 3, 5] # Istruzione di assegnamento
>>> a
[1, 2, 3, 5]
>>> x = (a = [1, 2, 3, 5]) # L'istruzione di assegnamento non è una espressione
      File "<stdin>", line 1
x = (a = [1, 2, 3, 5])
      ^
SyntaxError: invalid syntax
```

È possibile assegnare il medesimo oggetto a diverse etichette in un'unica istruzione:

```
>>> a = b = c = 1
```

e questa risulta equivalente alla sequenza dei singoli assegnamenti:

```
>>> c = 1
>>> b = c
>>> a = b
```

È anche possibile effettuare degli assegnamenti in parallelo, mediante lo *spacchettamento*:

```
>>> a, b, c = 1, 2, 3
>>> b
2
>>> a, b, c, d, e, f = 'python'
>>> a
'p'
>>> a, b, c, d = [1, 2, 3, 4]
>>> d
4
```

Parleremo in dettaglio dello spacchettamento nel prossimo capitolo, nella sezione dal titolo *Operazioni e funzioni built-in utilizzabili con gli oggetti iterabili*.

L'istruzione pass

La parola chiave `pass` è una istruzione semplice che non esegue nulla. Quando la si incontra viene saltata e l'esecuzione prosegue dall'istruzione successiva:

```
>>> pass # Istruzione 'pass', non esegue nulla
>>> a = (pass) # Non è una espressione
      File "<stdin>", line 1
```

```
a = (pass)
^
SyntaxError: invalid syntax
```

L'istruzione del

L'istruzione `del` cancella una o più etichette:

```
>>> a = b = [1, 2, 3]
>>> a is b
True
>>> del a # Cancella l'etichetta `a`, non l'oggetto
>>> a # L'etichetta non esiste più
Traceback (most recent call last):
...
NameError: name 'a' is not defined
>>> b # L'istruzione `del a` ha cancellato l'etichetta, non l'oggetto
[1, 2, 3]
>>> a = (del b) # Non è una espressione
      File "<stdin>", line 1
a = (del b) # Non è una espressione
^
SyntaxError: invalid syntax
```

L'istruzione return

L'istruzione `return`, da utilizzare solo all'interno di una funzione, serve per restituire esplicitamente al chiamante il risultato della funzione:

```
>>> def foo(x):
...     return x ** 2
...
>>> foo(4)
16
>>> def foo(x):
...     a = (return x ** 2) # Non è una espressione
      File "<stdin>", line 2
a = (return x ** 2)
^
SyntaxError: invalid syntax
```

Se una funzione non restituisce esplicitamente un oggetto al chiamante, allora viene restituito implicitamente l'oggetto `None`, unica istanza del tipo `NoneType`:

```
>>> def foo():
...     pass
...
>>> None is foo()
True
```

L'istruzione `import`

L'istruzione `import` serve per importare i moduli:

```
>>> import math
>>> math.pi
3.141592653589793
>>> m = (import math) # Non è una espressione
      File "<stdin>", line 1
m = (import math)
^
SyntaxError: invalid syntax
```

Vedremo in modo dettagliato l'istruzione `import` nel terzo capitolo, quando parleremo dei moduli.

L'istruzione `from`

L'istruzione `from` è utilizzata per importare degli attributi da un modulo:

```
>>> from math import pi, cos
>>> cos(pi) # Evito di scrivere `math.cos(math.pi)`
-1.0
>>> a = (from math import pi) # Non è una espressione
      File "<stdin>", line 1
a = (from math import pi)
^
SyntaxError: invalid syntax
```

L'istruzione `from` ha diverse varianti, che vedremo nel [Capitolo 4](#).

L'istruzione `assert`

L'istruzione `assert` solleva un'eccezione di tipo `AssertionError` se la sua espressione di test è valutata come `False`, altrimenti non fa nulla:

```
>>> assert 2*2 == 4
>>> assert 2*2 == 6
Traceback (most recent call last):
...
AssertionError
```

Vedremo più in dettaglio questa istruzione nel [Capitolo 5](#), quando parleremo delle eccezioni.

Istruzioni composte

L'istruzione composta è costituita da una o più parole chiave, chiamate *clausole*, a ciascuna delle quali è associata una *suite* di istruzioni. Le clausole

terminano con il segno dei due punti, dopo i quali inizia la suite:

```
>>> if 4 == 2 + 2: print('python'); print('openstack'); a = 33; print(a + 2)
python
openstack
35
```

Come ormai sappiamo, generalmente la suite si scrive in un blocco di codice indentato, scrivendo una istruzione per linea:

```
>>> if 4 == 2 + 2:
...     print('python')
...     print('openstack')
...     a = 33
...     print(a + 2)
...
python
openstack
35
```

Le clausole hanno tutte lo stesso livello di indentazione, come possiamo osservare dal seguente esempio, che mostra l'istruzione `if` e la sua clausola `else`:

```
>>> if 4 == 2 + 3:
...     print('if suite')
... else:
...     print('else suite')
...
else suite
```

In questa sezione mostreremo le modalità di utilizzo delle tre principali istruzioni composte: `if`, `for` e `while`.

L'istruzione `if`

In Python vi sono due strumenti che permettono di intraprendere azioni diverse sulla base del risultato di un test di verità. Il primo è l'espressione condizionale, della quale abbiamo già parlato, mentre il secondo è l'istruzione composta `if`, avente le clausole `elif` ed `else`. La sua forma generale è la seguente:

```
if <espressione di test n.1>:
<blocco di istruzioni n.1>
elif <espressione di test n.2>:
<blocco di istruzioni n.2> # Opzionale
else:
```

e questo è un esempio di utilizzo:

```
>>> s = 'python'
>>> if 'm' in s:
...     print('mmm')
... elif 'n' in s:
...     print('nnn')
... else:
...     print(s)
...
nnn
```

In Python non esiste una istruzione per le scelte multiple analoga alla *switch-case* tipica di altri linguaggi (vedi PEP-0275 e PEP-3103). Questa scelta è dovuta al fatto che è possibile ottenere in modo ovvio lo stesso risultato con l'istruzione `if` e, come già sappiamo, “there should be one - and preferably only one - obvious way to do it”.

È possibile ottenere lo *switch-case* anche con i dizionari, probabilmente in modo meno intuitivo rispetto alla corrispondente codifica con l'istruzione `if`:

```
>>> def switch(codice):
...     d = {0: len, 1: min, 2: max}
...     return d.get(codice, sum)
...
>>> switch(0)([1, 2, 3]) # len([1, 2, 3])
3
>>> switch(2)((5, 9, 1)) # max((5, 9, 1))
9
>>> switch(7)((5, 9, 1)) # default: sum((5, 9, 1))
15
```

L'istruzione `for`

L'istruzione `for` consente di eseguire delle iterazioni:

```
>>> for i in ('a', 'b'):
...     print(i)
...
a
b
>>> for i in [(1, 'uno'), (2, 'due'), (3, 'tre')]:
...     print(i)
...
(1, 'uno')
```

```
(2, 'due')
(3, 'tre')
```

Gli elementi dell'*oggetto iterabile* possono anche essere *spacchettati*:

```
>>> for i, j in [(1, 'uno'), (2, 'due'), (3, 'tre')]:
...     print(i, j)
...
1 uno
2 due
3 tre
```

Quando all'interno di un ciclo `for` viene eseguita l'istruzione semplice `continue`, le restanti istruzioni del blocco vengono saltate e l'esecuzione riprende dalla linea contenente la parola chiave `for`:

```
>>> for i in range(20):
...     if i % 2 == 0: # Se il numero è pari
...         continue
...     print(i, end=' ')
...
1 3 5 7 9 11 13 15 17 19
```

L'istruzione semplice `break` fa terminare l'esecuzione del ciclo `for`:

```
>>> for i in range(20):
...     if i == 13:
...         break
...     print(i, end=' ')
...
0 1 2 3 4 5 6 7 8 9 10 11 12
```

È possibile anche utilizzare la clausola `else`, che viene eseguita se si esce dal ciclo senza incontrare l'istruzione `break`:

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

Nella sezione *Oggetti iterabili, iteratori e contesto di iterazione*, al termine di questo capitolo, descriveremo nel dettaglio i meccanismi che stanno alla base

del protocollo di iterazione.

L'istruzione `while`

L'istruzione `while` consente di eseguire dei cicli infiniti:

```
>>> while True:
...   x = input('Digita qualcosa (esci con Ctl-D su Unix, Ctl-Z+Return su Win): ')
...   print('Hai digitato', x)
...
Digita qualcosa (esci con Ctl-D su Unix, Ctl-Z+Return su Win): python
Hai digitato python
Digita qualcosa (esci con Ctl-D su Unix, Ctl-Z+Return su Win): c++
Hai digitato c++
```

Possiamo utilizzare `continue`, `break` e la clausola `else` nello stesso modo usato per l'istruzione `for`.

I moduli come contenitori di istruzioni

Abbiamo appena visto che le espressioni sono contenute nelle istruzioni. Queste, a loro volta, sono contenute all'interno di *moduli Python*. I moduli Python sono dei file contenenti zero (un file vuoto, il modulo più banale) o più istruzioni Python. Questi vengono importati utilizzando l'istruzione `import`, e a loro volta possono importare altri moduli.

Un modulo Python, per poter essere importato, deve necessariamente essere un file con suffisso `.py`, ma quando si scrive l'istruzione `import` il nome del modulo va scritto senza suffisso. Consideriamo, ad esempio, il seguente file *foo*:

```
$ cat foo
print('Ciao, quanto è bello Python :)')
```

Possiamo eseguirlo:

```
$ python foo
Ciao, quanto è bello Python :)
```

ma non importarlo:

```
>>> import foo # Il file foo non ha suffisso '.py', non possiamo quindi importarlo come modulo
Traceback (most recent call last):
...
ImportError: No module named 'foo'
```

Quando importiamo un modulo, questo viene eseguito:

```
$ mv foo foo.py # Rinominiamo il file aggiungendo '.py'. Ora possiamo importarlo
$ python -c "import foo"
Ciao, quando è bello Python :)
```

L'istruzione `from` consente di copiare gli attributi di un modulo, ovvero le etichette definite al suo interno:

```
$ cat mymodule.py
mylist = [1, 2, 3]
$ python -c "from mymodule import mylist; print(mylist)"
[1, 2, 3]
$ python -c "from mymodule import mylist; mylist.append(4); print(mylist)"
[1, 2, 3, 4]
```

Abbiamo detto, all'inizio di questa sezione, che le istruzioni sono contenute nei moduli. Allora ci saremo forse chiesti in quale modulo sono contenute le istruzioni eseguite in modo interattivo. L'etichetta globale `__name__` ci dà la risposta. Questa, infatti, fa riferimento al nome del modulo nel quale è utilizzata:

```
>>> __name__
'__main__'
```

Quindi il codice eseguito in modo interattivo risiede all'interno di un modulo chiamato `__main__`:

```
>>> b = 100
>>> import __main__
>>> __main__.b
100
>>> c
Traceback (most recent call last):
...
NameError: name 'c' is not defined
>>> __main__.c = 50
>>> c
50
```

Anche un file *mymodule.py*, quando eseguito direttamente da linea di comando, viene caricato in memoria con il nome `__main__` anziché con il nome *mymodule*. Consideriamo, ad esempio, il seguente file *mymodule.py*:

```
$ cat mymodule.py
print(__name__)
```



```
a = 100
import __main__
print(__main__.a)
__main__.a = 200
print(a)
```

Ecco cosa accade quando lo eseguiamo da linea di comando:

```
$ python mymodule.py
__main__
100
200
```

L'etichetta globale `__file__`, usata all'interno di un modulo, fa riferimento al nome del file:

```
$ cat mymodule.py
print(__file__)
$ python mymodule.py
mymodule.py
```

La nostra introduzione sui moduli finisce qui; per una approfondita discussione rimandiamo alla sezione *Moduli* del Capitolo 4.

La Python Virtual Machine

L'installazione di Python, in realtà, non contiene solo un interprete, ma anche un *compilatore*. Quando un programma viene mandato in esecuzione, Python compila tutti i moduli importati che non hanno subito modifiche dall'ultima volta in cui sono stati compilati. Questo processo genera una nuova rappresentazione (di basso livello) del codice, detta *bytecode*, la quale viene letta e mandata in esecuzione dalla *Python Virtual Machine* (PVM). Non è quindi il codice Python a venire eseguito, ma il codice in bytecode, per cui il vero interprete, in realtà, è la PVM, la quale esegue le istruzioni bytecode una alla volta.

Di seguito, un esempio di istruzioni bytecode relative a una funzione che restituisce il quadrato di un numero:

```
>>> def square(x):
...     return x * x
...
>>> import dis
>>> dis.dis(square) # Mostra le istruzioni in bytecode della funzione `square()`
2 0 LOAD_FAST 0 (x)
3 LOAD_FAST 0 (x)
6 BINARY_MULTIPLY
7 RETURN_VALUE
```

NOTA

Per maggiori informazioni sul significato delle istruzioni in bytecode, possiamo consultare la documentazione online del modulo `dis`: <http://docs.python.org/py3k/library/dis.html>. Teniamo presente che tutti i dettagli implementativi discussi in questo libro sono relativi a CPython.

Quando un modulo viene importato, la sua versione compilata, se non espresso diversamente, viene salvata nel *file-system* in modo da poter essere riutilizzata durante le successive esecuzioni del programma, senza che siano necessarie ulteriori compilazioni (vedi PEP-3147):

```
$ # Creiamo un modulo Python
$ echo "print(__name__)" > m.py # L'istruzione `print(__name__)` stampa il nome del modulo
```

```

$ ls
m.py
$ # Importiamo il modulo Python
$ python -c "import m" # Il codice contenuto nel modulo viene eseguito
m
$ ls
m.py __pycache__
$ # È stata creata la directory __pycache__
$ ls __pycache__
m.cpython-33.pyc
$ # Nella directory __pycache__ è stata salvata la versione compilata di m.py
$ # La versione compilata è stata salvata al seguente orario
$ ls -l --time-style=full-iso __pycache__/m.cpython-33.pyc | cut -f7 -d' '
11:25:02.000000000
$ # Non abbiamo modificato m.py, quindi non verrà ricompilato se importato
$ python -c "import m"
m
$ ls -l --time-style=full-iso __pycache__/m.cpython-34.pyc | cut -f7 -d' '
11:25:02.000000000
$ # È stata quindi eseguita la versione compilata con il primo `import`
$ # Modifichiamo la data di ultimo accesso di m.py
$ touch m.py
$ # Se adesso importiamo il modulo, questo verrà ricompilato
$ python -c "import m"
m
$ ls -l --time-style=full-iso __pycache__/m.cpython-34.pyc | cut -f7 -d' '
11:28:01.000000000

```

I file compilati in bytecode possono essere eseguiti direttamente:

```

$ python __pycache__/m.cpython-34.pyc
__main__

```

NOTA

Nell'esempio precedente abbiamo utilizzato vari comandi tipici delle shell Unix. Vediamo il significato di qualcuno di questi. Il carattere `#`, come possiamo immaginare, viene utilizzato per iniziare un commento. L'istruzione `echo "print(__name__)" > m.py` scrive la stringa `print(__name__)` sul file *m.py* (se il file non esiste lo crea, se esiste lo sovrascrive). Il comando `ls` elenca informazioni sui file e mostra il contenuto delle directory. Le spiegazioni dei restanti comandi e delle opzioni sono riportate nell'Appendice A.

I moduli Python importati hanno gli attributi `__file__` e `__cached__`, i quali sono delle stringhe che rappresentano, rispettivamente, il nome del file sorgente e quello del file importato:

```
>>> import m
m
>>> m.__file__ # File sorgente
'./m.py'
>>> m.__cached__ # File importato
'./__pycache__/m.cpython-34.pyc'
```

NOTA

Vedremo, nella sezione *I moduli* del [Capitolo 3](#), che alcuni moduli (*built-in* e *namespace packages*) non sono riconducibili a un file, e quindi non hanno gli attributi `__file__` e `__cached__`.

Il *tag* che identifica l'implementazione dell'interprete e la versione di Python è dato da `imp.get_tag()`:

```
>>> import imp
>>> imp.get_tag() # Restituisce la stringa 'implementazione-versione'
'cpython-34'
```

Il modulo `imp` ci consente anche di ottenere il nome del file sorgente a partire dal nome del file importato e viceversa:

```
>>> import os
>>> os.__file__
'/usr/local/lib/python3.4/os.py'
>>> imp.source_from_cache(os.__cached__)
'/usr/local/lib/python3.4/os.py'
>>> imp.cache_from_source(os.__file__)
'/usr/local/lib/python3.4/__pycache__/os.cpython-34.pyc'
```

Quando un modulo Python viene eseguito come script, il suo bytecode non viene salvato sul disco:

```
$ rm __pycache__ -r
$ ls
m.py
$ python m.py # La versione compilata non viene salvata
__main__
$ ls
m.py
```

NOTA

Il comando `rm` rimuove un file o una directory dal file-system. Nel caso di

una directory, è necessario utilizzare l'opzione `-r`.

Possiamo, però, chiedere esplicitamente che venga salvato. Per far ciò si utilizza lo switch `-m` e si passa all'interprete il nome del modulo piuttosto che quello del file:

```
$ python -m m # Passo il nome del modulo (m), non quello del file (m.py)
__main__
$ ls
m.py __pycache__
$ ls __pycache__
m.cpython-34.pyc
```

Se in combinazione con lo switch `-m` si utilizza anche quello `-O`, allora viene generato del bytecode ottimizzato (vengono rimosse le istruzioni `assert`), il quale viene salvato in un file con suffisso `.pyo`:

```
$ python -O -m m # Generiamo un file '.pyo' con bytecode ottimizzato
__main__
$ # Non vi sono istruzioni 'assert', quindi versione '.pyo' e '.pyc' non differiscono
$ diff __pycache__/m.cpython-34.pyc __pycache__/m.cpython-34.pyo
$ # Aggiungiamo una istruzione 'assert'
$ echo "assert True" >> m.py
$ python -m m # Generiamo il '.pyc'
__main__
$ python -O -m m # Generiamo il '.pyo'
__main__
$ # Nel '.pyo' è stata rimossa l'istruzione 'assert', per cui differiscono
$ diff __pycache__/m.cpython-34.pyc __pycache__/m.cpython-34.pyo
Binary files __pycache__/m.cpython-34.pyc and __pycache__/m.cpython-34.pyo differ
$ # I file '.pyo' possono essere eseguiti direttamente
$ python __pycache__/m.cpython-34.pyo
__main__
```

NOTA

Dati due file binari *file_A* e *file_B*, il comando `diff file_A file_B` non mostra alcun output se i due file sono identici, altrimenti indica che essi differiscono.

Possiamo anche scegliere di non salvare nel file-system le versioni compilate dei moduli importati. Per far ciò possiamo utilizzare lo switch `-B`:

```
$ echo "print(__name__)" > m.py
$ python -B -c "import m" # Utilizzo dello switch '-B'
m
$ ls
```

m.py

oppure possiamo assegnare alla variabile d'ambiente `PYTHONDONTWRITEBYTECODE` il valore `true`:

```
$ PYTHONDONTWRITEBYTECODE=true # Impostazione della variabile PYTHONDONTWRITEBYTECODE
$ export PYTHONDONTWRITEBYTECODE
$ python -c "import m"
m
$ ls
m.py
```

oppure possiamo farlo quando il programma è in esecuzione, ponendo `sys.dont_write_bytecode = True`:

```
$ python -c "import sys; sys.dont_write_bytecode=True; import m"
m
$ ls
m.py
```

Etichette e oggetti

Spesso si dice che ogni cosa in Python è un oggetto. In realtà, come abbiamo più volte detto, gli oggetti non fanno parte del codice, ma vengono da esso generati quando il programma è in esecuzione. In questa sezione daremo una definizione formale di oggetto, raccogliendo quanto seminato nella prima parte del libro. Vedremo, inoltre, in modo approfondito qual è il legame tra gli oggetti e le etichette, poiché questo potrebbe non essere immediatamente evidente a chi ha solo esperienze di programmazione con linguaggi a tipizzazione statica.

Definizione di oggetto

Gli *oggetti* in Python sono delle strutture dati generate dal codice in esecuzione, alle quali il codice fa riferimento. Precisamente, sono le espressioni che generano o fanno riferimento agli oggetti.

Finché servono, gli oggetti vengono tenuti nella memoria del computer in modo da poter essere richiamati all'occorrenza; nel momento in cui, però, non vi è più alcuna etichetta che fa riferimento a essi, un particolare meccanismo, chiamato *garbage collection*, provvede a liberare la memoria da essi occupata.

NOTA

Possiamo gestire il meccanismo del garbage collection utilizzando il modulo `gc`, che fornisce una interfaccia per il *garbage collector*: <http://docs.python.org/3/library/gc.html>.

Gli oggetti hanno un *tipo* e una *identità*. Possiamo ottenere il tipo di un oggetto sia mediante la classe built-in `type` sia mediante l'attributo `__class__` dell'oggetto:

```
>>> a = 22
>>> a.__class__, type(a)
(<class 'int'>, <class 'int'>)
```

L'identità di un oggetto, che in CPython corrisponde al suo indirizzo in

memoria, viene restituita dalla funzione built-in `id()`:

```
>>> id([1, 2, 3]) # Identificativo dell'oggetto che in memoria rappresenta la lista [1, 2, 3]
3072196748
>>> id(22) # Identificativo dell'oggetto che in memoria rappresenta il numero intero 22
136597616
>>> id(id) # Identificativo dell'oggetto che in memoria rappresenta la funzione built-in `id()`
3073367564
```

Osserviamo, inoltre, come due oggetti distinti abbiano identità distinta pur avendo lo stesso valore:

```
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2, 3]
>>> id(l1), id(l2)
(3072196908, 3072197932)
```

NOTA

Quando l'interprete incontra i letterali dei tipi immutabili, per questioni di ottimizzazione (indipendentemente dall'implementazione) predilige il riutilizzo di oggetti già esistenti piuttosto che la creazione di nuovi, specie se si tratta di oggetti che occupano poco spazio in memoria:

```
>>> a = 'python'
>>> b = 'python'
>>> a is b
True
>>> a = 'Supercalifragilisticexpialidocious!'
>>> b = 'Supercalifragilisticexpialidocious!'
>>> a is b
False
```

Si badi che il fatto che ogni struttura dati generata dal codice Python in esecuzione sia un oggetto non significa affatto che l'unico paradigma di programmazione supportato sia quello a oggetti. Come abbiamo detto nella sezione *Lo stato dell'arte*, Python è un linguaggio *multi-paradigma*.

Il legame tra le etichette e gli oggetti

Come ormai sappiamo, in Python non vi sono dichiarazioni, e il tipo degli oggetti viene stabilito quando il programma è in esecuzione. Forse allora ci chiediamo quale sia il ruolo delle etichette, visto che possiamo assegnare oggetti di tipo diverso alla stessa etichetta:


```
>>> var = 44 # Creiamo un oggetto di tipo `int` e lo assegniamo alla etichetta `var`
>>> var, type(var), id(var)
(44, <class 'int'>, 136597840)
>>> var = 99 # L'etichetta `var` è legata a un nuovo oggetto, ancora di tipo `int`
>>> var, type(var), id(var)
(99, <class 'int'>, 136598720)
>>> var = 'python' # L'etichetta `var` è legata a un oggetto di tipo stringa
>>> var, type(var), id(var)
('python', <class 'str'>, 3074491872)
```

Il punto è che una *etichetta* è semplicemente un identificativo tramite il quale fare riferimento a un oggetto. Ciò significa che il primo assegnamento ha legato l'etichetta `var` all'oggetto rappresentativo del numero 44, di tipo `int`, come mostrato in [Figura 1.1](#).



Figura 1.1 - Assegnamento dell'etichetta `var` a un numero intero di valore 44.

Il secondo assegnamento ha legato `var` a un nuovo oggetto, rappresentativo del numero 99 e ancora di tipo `int`, e non esiste più alcun legame tra `var` e l'oggetto rappresentativo del numero 44. Inoltre quest'ultimo non ha alcuna etichetta che fa riferimento a esso, per cui non può essere utilizzato in alcun modo, e Python, pertanto, libera la memoria da esso occupata, come mostrato in [Figura 1.2](#).



Figura 1.2 - Assegnamento dell'etichetta `var` a un nuovo numero intero, di valore 99. La memoria viene liberata dal 44.

Infine, il terzo assegnamento ha legato `var` all'oggetto rappresentativo della stringa `'python'`, come mostrato in [Figura 1.3](#).



Figura 1.3 - Assegnamento di un nuovo oggetto di tipo `str` all'etichetta `var`.

Nel prosieguo del libro, per agevolare l'esposizione, non parleremo più di oggetto rappresentativo di un numero, di una stringa o di altro, ma diremo semplicemente oggetto di tipo `int`, oggetto di tipo `str` ecc. Inoltre, utilizzeremo il

termine *referimento* per indicare il legame tra l'etichetta e l'oggetto:

```
>>> var = [1, 2, 3] # È stato creato un oggetto lista, assegnato a `var`
>>> id(var) # L'etichetta `var` è un *referimento* all'oggetto con id 3073291148
3073291148
>>> print # `print` è una etichetta che fa riferimento a una funzione built-in
<built-in function print>
>>> print = 99 # Infatti posso assegnare all'etichetta `print` un altro oggetto
>>> print
99
>>> class Foo:
...     pass
...
>>> Foo # `Foo` è una etichetta
<class '__main__.Foo'>
>>> Foo = 33
>>> Foo
33
```

Una etichetta può essere utilizzata solo dopo essere stata assegnata a un oggetto:

```
>>> print(a) # Proviamo a stampare una etichetta che non abbiamo definito
Traceback (most recent call last):
...
NameError: name 'a' is not defined
```

NOTA

Come possiamo osservare da questo esempio, il termine utilizzato dall'interprete per indicare ciò che noi abbiamo chiamato etichetta è *name*. Anche la maggior parte dei libri e della documentazione in lingua inglese utilizza il termine *name*.

Un'operazione di assegnamento a una etichetta già esistente non modifica mai l'oggetto a essa precedentemente assegnato:

```
>>> var = ('a', 'b', 'c')
>>> id(var)
3073908940
>>> var = ('a', 'b', 'c') + ('d',) # Creiamo una nuova tupla e la assegniamo a `var`
>>> id(var) # Infatti `var` fa riferimento a un oggetto diverso dal precedente
3073909380
```

Questo è sempre vero, anche se l'oggetto è mutabile. Consideriamo, ad esempio, la seguente lista:

```
>>> var = [1, 2, 3]
>>> var, type(var), id(var)
([1, 2, 3], <class 'list'>, 3073876076)
```

della quale riportiamo una illustrazione in [Figura 1.4](#).



Figura 1.4 - Rappresentazione del riferimento tra l'etichetta `var` e la lista `[1, 2, 3]`.

Se facciamo l'assegnamento `var = [1, 2, 3] + [4]`, nonostante la lista sia mutabile, l'oggetto con identificativo `3073876076` non viene modificato. Questa istruzione ha il solo effetto di creare un nuovo oggetto di tipo lista e assegnarlo all'etichetta `'var'`:

```
>>> var = [1, 2, 3] + [4]
>>> var, type(var), id(var) # L'id dell'oggetto è diverso dal precedente
([1, 2, 3, 4], <class 'list'>, 3073876972)
```

La precedente lista è ora inutilizzabile poiché non vi sono più etichette che fanno riferimento a essa, per cui Python libera la memoria occupata, come schematizzato in [Figura 1.5](#).

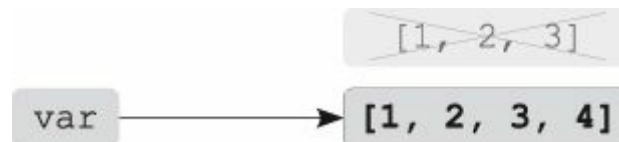


Figura 1.5 - L'etichetta `var` fa riferimento alla lista `[1, 2, 3, 4]`. La memoria viene liberata dalla lista `[1, 2, 3]`.

L'effetto dell'assegnamento esplicito è, quindi, sempre e solo quello di creare un legame tra l'etichetta e l'oggetto alla destra del simbolo di uguale.

Se ci fa comodo avere una rappresentazione visuale di ciò che avviene quando all'interno di una espressione è presente una etichetta, possiamo pensare (e non commetteremmo alcun errore logico) che Python sostituisca l'etichetta con l'oggetto a cui essa fa riferimento.

La creazione dei riferimenti tra etichette e oggetti avviene nella stessa maniera per ogni modalità di assegnamento, quindi anche per quella *implicita*, come, ad esempio, nel caso delle etichette assegnate implicitamente nelle istruzioni `for`:

```
>>> mylist = ['a', 'b', 'c']
>>> [id(item) for item in mylist]
[3073820736, 3074319456, 3074282848]
>>> for item in mylist:
```

```
... print(id(item))
...
3073820736
3074319456
3074282848
```

dove l'etichetta `item` a ogni iterazione si riferisce al corrispondente oggetto presente nella lista.

Un altro caso di assegnazione implicita è il passaggio degli argomenti a una funzione, dove le etichette utilizzate come parametri fanno riferimento ai rispettivi oggetti passati come argomenti, e non a una loro copia:

```
>>> def foo(par1, par2):
...     print(par1, id(par1), sep=':')
...     print(par2, id(par2), sep=':')
...
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> id(a), id(b)
(3074386092, 3074386124)
>>> foo(a, b)
[1, 2, 3]:3074386092
[4, 5, 6]:3074386124
```

Infatti, quando è stata fatta la chiamata `foo(a, b)`, sono avvenuti gli assegnamenti impliciti `par1=a` e `par2=b`, e possiamo pensare che `a` venga sostituita con l'oggetto a cui fa riferimento, così come `b`.

Anche la definizione di una funzione dà luogo a un assegnamento implicito. L'istruzione `def` crea un oggetto funzione e lo assegna all'etichetta interposta tra la `def` e le parentesi tonde:

```
>>> def foo():
...     pass
...
>>> foo
<function foo at 0xb72da4f4>
>>> foo = 100 # `foo` è una etichetta che fa riferimento all'intero 100
>>> foo
100
```

Anche con l'*indicizzazione* si ottiene lo stesso comportamento. Ad esempio, nel seguente codice possiamo pensare che `mylist[0]`, `mylist[1]` e `mylist[2]` vengano sostituiti con gli oggetti a cui fanno riferimento:

```
>>> item = mylist[0]
```

```
>>> id(item)
3073820736
>>> item = mylist[1]
>>> id(item)
3074319456
>>> item = mylist[2]
>>> id(item)
3074282848
```

Oggetti mutabili e immutabili

Date due etichette *a* e *b* che fanno riferimento al medesimo oggetto, cosa accade se facciamo un assegnamento a una delle due? Spero che nessuno si aspetti che venga modificato l'oggetto al quale fanno riferimento. Abbiamo detto, infatti, che non è possibile modificare un oggetto con un assegnamento. Ciò che accade è che l'etichetta farà riferimento al nuovo oggetto che le verrà assegnato. Consideriamo il seguente esempio:

```
>>> a = b = [1, 2, 3]
>>> a, b
([1, 2, 3], [1, 2, 3])
>>> id(a), id(b)
(3074330092, 3074330092)
```

Le due etichette *a* e *b* fanno riferimento al medesimo oggetto, come mostrato in [Figura 1.6](#).



Figura 1.6 - Le etichette *a* e *b* fanno riferimento al medesimo oggetto, avente identità 3074330092.

Se ora facciamo l'assegnamento *a* = [1, 2, 3, 4], l'oggetto con identità 3074330092 non verrà modificato:

```
>>> a = [1, 2, 3, 4]
>>> a, id(a)
([1, 2, 3, 4], 3072863756)
>>> b, id(b)
([1, 2, 3], 3074330092)
```

Infatti, è stato creato un nuovo oggetto, rappresentativo della lista [1, 2, 3, 4], che è stato assegnato all'etichetta *a*, mentre l'etichetta *b* ha continuato a fare riferimento al medesimo oggetto di prima, come illustrato in [Figura 1.7](#).

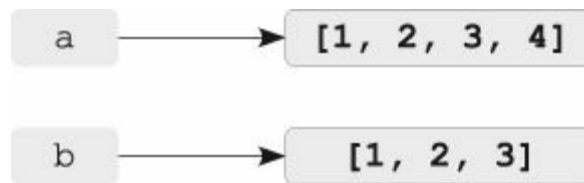


Figura 1.7 - L'etichetta *a* fa riferimento a un nuovo oggetto, mentre *b* continua a fare riferimento all'oggetto precedente.

A questo punto ci saremo forse già posti la seguente domanda: “come possiamo modificare un oggetto, visto che non è possibile farlo con un assegnamento né implicito né esplicito?”. La risposta ci consentirà di capire meglio la differenza tra oggetti mutabili e immutabili.

Modifica di oggetti mutabili

Gli oggetti mutabili possono essere modificati solo in tre modi: tramite i loro metodi, tramite gli *augmented assignment* e tramite l'istruzione `del`. Ad esempio, i metodi `list.append()` e `list.sort()` di una lista modificano l'oggetto:

```
>>> mylist = [3, 1, 2]
>>> id(mylist)
3072824204
>>> mylist.append(4) # Modifica l'oggetto
>>> id(mylist) # 'mylist' fa ancora riferimento al precedente oggetto
3072824204
>>> mylist # Il quale è stato modificato
[3, 1, 2, 4]
>>> mylist.sort() # Anche il metodo 'list.sort()' modifica l'oggetto
>>> mylist
[1, 2, 3, 4]
>>> id(mylist)
3072824204
```

Gli *augmented assignment*, come specificato nella PEP-0203, modificano anch'essi l'oggetto:

```
>>> mylist += [5]
>>> mylist, id(mylist)
([1, 2, 3, 4, 5], 3072824204)
>>> mylist *= 2
>>> mylist, id(mylist)
([1, 2, 3, 4, 5, 1, 2, 3, 4, 5], 3072824204)
```

Infine, l'istruzione `del`, utilizzata in un contesto di indicizzazione o slicing, consente di eliminare degli elementi da un oggetto contenitore, modificando così l'oggetto:

```
>>> del mylist[3:]
>>> mylist
```

```
[1, 2, 3]
>>> id(mylist)
3072824204
```

Se, invece, è applicata a una semplice etichetta, ha solo l'effetto di eliminarla dal namespace:

```
>>> a = b = [1, 2, 3]
>>> del a # Elimina l'etichetta, non l'oggetto a cui essa fa riferimento
>>> b # Infatti l'oggetto esiste ancora ed è legato a `b`
[1, 2, 3]
```

Quando un oggetto non ha più etichette che vi fanno riferimento, allora Python libera la memoria da esso occupata:

```
>>> class Foo():
...     def __del__(self):
...         """Questo metodo è invocato quando l'istanza viene distrutta"""
...         print("Sto liberando la memoria...")
...
>>> a = b = Foo()
>>> id(a), id(b) # Le due etichette fanno riferimento allo stesso oggetto
(3072491148, 3072491148)
>>> del a # C'è ancora l'etichetta `b` che fa riferimento all'oggetto
>>> del b # Adesso la memoria viene liberata dall'oggetto
Sto liberando la memoria...
```

NOTA

Gli oggetti in CPython hanno internamente due campi: un *descrittore di tipo*, utilizzato per determinare il tipo dell'oggetto, e un contatore dei riferimenti (*reference count*), utilizzato per capire quando l'oggetto non ha più etichette che fanno riferimento a esso e può quindi essere eliminato dalla memoria. Per maggiori dettagli: <http://docs.python.org/py3k/c-api/intro.html>.

Proviamo a chiarire questo concetto con delle illustrazioni. Inizialmente sono state create due etichette *a* e *b* che facevano riferimento al medesimo oggetto, come mostrato in Figura 1.8.



Figura 1.8 - Le due etichette a e b fanno riferimento al medesimo oggetto.

L'istruzione `del a` ha cancellato l'etichetta `a`, ma la memoria non è stata liberata dall'oggetto poiché c'era ancora un'etichetta che faceva riferimento a esso, come mostrato in [Figura 1.9](#).



Figura 1.9 - L'istruzione `del a` ha eliminato l'etichetta `a`, ma non l'oggetto al quale essa faceva riferimento.

Quando, invece, è stata cancellata anche l'etichetta `b`, l'oggetto non aveva più etichette che vi facessero riferimento, per cui, essendo inutilizzabile, Python ha eseguito il metodo `__del__()` e successivamente ha liberato la memoria occupata dall'oggetto.

Immutabilità

Gli oggetti immutabili non hanno metodi che consentono di modificarli, e neppure gli augmented assignment possono farlo:

```
>>> t = (1, 2, 3)
>>> id(t)
3073834516
>>> t += (4, 5)
>>> id(t) # È stato creato un nuovo oggetto e assegnato a `t`
3074521676
```

Non c'è quindi alcun modo per modificarli. Ad esempio, data la seguente stringa:

```
>>> s = "Mi sono innamorato... "
>>> id(s)
3073911456
```

se volessimo estenderla non potremmo farlo. Possiamo solo creare un nuovo oggetto e assegnarlo all'etichetta `s`:

```
>>> s = s + "di Python"
>>> s
'Mi sono innamorato... di Python'
>>> id(s) # L'etichetta `s` fa riferimento a un nuovo oggetto
3073858656
```


Oggetti immutabili contenenti oggetti mutabili

Consideriamo ora il caso di un oggetto immutabile contenente degli oggetti mutabili, come quello di una tupla contenente un set e una lista:

```
>>> t = ({'a', 'b'}, [1, 2, 3], 'python')
>>> id(t[0]), id(t[1]), id(t[2])
(3072619996, 3072561292, 3073177088)
```

Gli elementi di un oggetto contenitore sono semplicemente dei riferimenti ai corrispondenti oggetti, come mostrato in [Figura 1.10](#).

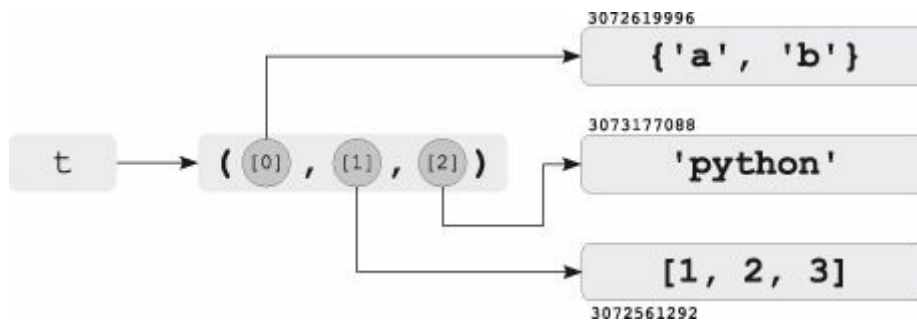


Figura 1.10 - Oggetto immutabile contenente oggetti mutabili.

L'immutabilità consiste nel non poter modificare i riferimenti della tupla, ovvero nel non poter effettuare un assegnamento del seguente tipo:

```
>>> t[0] = {'a', 'b', 'c'}
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Quindi non è possibile che `t[0]` faccia riferimento a un oggetto diverso dal set con identità 3072619996. Questo, però, non significa che l'oggetto a cui `t[0]` fa riferimento non possa essere modificato:

```
>>> t[0].add('c') # Aggiungo l'elemento 'c' al set con identità 3072619996
>>> t[1].append(4) Aggiungo l'elemento '4' alla lista con identità 3072561292
>>> t
({'a', 'c', 'b'}, [1, 2, 3, 4], 'python')
```

Il risultato di `t[0].add('c')` e `t[1].append(4)` è illustrato in [Figura 1.11](#), dove è mostrato come gli oggetti a cui `t[0]` e `t[1]` fanno riferimento sono stati modificati.

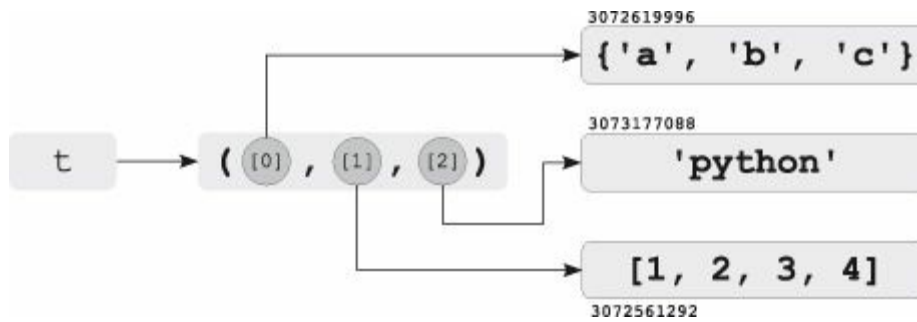


Figura 1.11 -Gli oggetti mutabili contenuti in un oggetto immutabile possono essere modificati.

Gli attributi sono etichette

Quando utilizziamo la notazione `foo.attributo`, sia `foo` sia `attributo` sono *etichette*, ciascuna delle quali fa riferimento al corrispondente oggetto:

```
>>> class Foo:
...     def foomethod(self):
...         print('python')
...
>>> Foo.pippo = Foo.foomethod
>>> Foo.pippo is Foo.foomethod # I due attributi fanno riferimento al medesimo oggetto
True
>>> f = Foo()
>>> f.pippo()
python
>>> baudo = f.pippo
>>> baudo() # `baudo` e `f.pippo` fanno riferimento allo stesso oggetto
python
```

Quindi, se parliamo dell'attributo `path` del modulo `sys`, con il termine “attributo” intendiamo “etichetta”. L'attributo, essendo sinonimo di etichetta, fa quindi riferimento a un oggetto, che nel caso di `sys.path` è una lista:

```
>>> import sys
>>> type(sys.path)
<class 'list'>
```

Etichette e nomi

Nella sezione *Gli elementi del codice Python* abbiamo detto che, per creare e rappresentare facilmente le istanze dei tipi del core data type, sono state definite delle forme testuali simboliche, chiamate *letterali*. Queste consentono di determinare visivamente sia il tipo sia il valore delle istanze:

```
>>> a = 99 # Oggetto creato tramite un letterale
>>> b = [1, 2, 3] # Oggetto creato tramite un letterale
>>> c = 100 # Oggetto creato tramite un letterale
>>> d = {'nome': 'Callisto', 'assunzione': (9, 3, 2008)} # Tramite letterale
```

```
>>> s = 'python' # Oggetto creato tramite un letterale
>>> d # La shell interattiva mostra la rappresentazione dell'oggetto
{'nome': 'Callisto', 'assunzione': (9, 3, 2008)}
```

Per gli altri tipi di oggetto il linguaggio non definisce una rappresentazione letterale, anche perché solitamente non è possibile farlo, visto che il concetto di *valore* non è immediato come per le istanze del core data type. Per essi è necessario, quindi, definire degli attributi che consentano di distinguere l'oggetto rispetto ad altri oggetti dello stesso tipo. Il più banale elemento di distinzione a cui possiamo pensare è quello che utilizziamo nella quotidianità per distinguere tra loro le persone: il *nome*. Per gli oggetti come le classi, le funzioni e i metodi, il nome viene assegnato all'attributo `__name__` al momento della loro creazione, sulla base del nome dell'etichetta utilizzata per far riferimento all'oggetto (per le funzioni e le classi) o del nome del file (per i moduli):

```
>>> def adder(x, y):
...     return x + y
...
>>> adder
<function adder at 0xb7258adc>
>>> adder.__name__
'adder'
>>> class Foo:
...     pass
...
>>> Foo.__name__
'Foo'
>>> import encodings
>>> encodings
<module 'encodings' from '/usr/local/lib/python3.4/encodings/__init__.py'>
>>> encodings.__name__
'encodings'
```

Se un oggetto ha un attributo designato a rappresentarne il nome, diciamo che il *nome dell'oggetto* è la stringa a cui fa riferimento questo attributo. Questo, quando esiste, non è sempre `__name__`. Per i file, ad esempio, viene utilizzato un attributo più esplicito (senza underscore), l'attributo `name`, il quale fa riferimento alla stringa passata a `open()` come primo argomento:

```
>>> a = 'nome del file'
>>> b = 'nome del file'
>>> a is b
False
>>> f = open(a, 'w')
>>> f.name
```

```
'nome del file'
>>> f.name is a
True
```

Il *nome di una etichetta* è la stringa costruita sulla base dell'etichetta: ad esempio, il nome dell'etichetta `foo` è la stringa `'foo'` e quello dell'attributo `path` del modulo `os` è la stringa `'path'`. La funzione built-in `dir()`, quando chiamata senza argomenti, restituisce una lista contenente i nomi delle etichette nello scope corrente, altrimenti, se le viene passato un oggetto, abbiamo visto, nelle precedenti sezioni, che essa restituisce i nomi degli attributi più significativi di quell'oggetto:

```
>>> def foo():
...     a = 33
...     mylist = [1, 2, 3]
...     print(dir())# Stampa la lista dei nomi delle etichette definite nello scope corrente
...
>>> foo() # Stampa la lista dei nomi delle etichette definite in `foo`
['a', 'mylist']
>>> import os
>>> dir(os) # Restituisce la lista dei nomi degli attributi del modulo `os`
['_CLD_CONTINUED', '_CLD_DUMPED', ..., 'path', ..., 'write', 'writev']
>>> 'path' in dir(os) # Il modulo `os` ha un attributo di nome 'path'?
True
```

Quindi il modulo `os` ha l'attributo `path` e il nome di questo attributo è la stringa `'path'`. Il nome dell'oggetto a cui l'attributo `path` di `os` fa riferimento, nei sistemi Unix-like, è la stringa `'posixpath'`:

```
>>> from os import path
>>> path.__name__ # Su Linux
'posixpath'
```

NOTA

Nell'ultima sezione di questo capitolo parleremo del modulo `os`, e vedremo che nei sistemi Windows `os.path` fa invece riferimento al modulo `ntpath`:

```
>>> from os import path
>>> path.__name__ # Su Windows
'ntpath'
```

Un oggetto non sempre ha un nome (le istanze del core data type, ad esempio,

non lo hanno), mentre una etichetta sì, perché questo è dato dall'etichetta sotto forma di stringa. Il nome di un oggetto, quindi, non ha nulla a che vedere con quello delle etichette che a esso fanno riferimento, anche perché, se ci pensiamo, più etichette possono fare riferimento allo stesso oggetto. L'unico legame tra i due è che, talvolta (come per le classi, le funzioni e i moduli), il nome dell'etichetta viene utilizzato al momento della creazione dell'oggetto per assegnare a questo il nome:

```
>>> def foo():
...     pass
...
>>> foo.__name__ # Nome dell'oggetto di tipo funzione a cui l'etichetta `foo` fa riferimento
'foo'
>>> 'foo' in dir() # In questo caso il nome dell'etichetta e quello dell'oggetto coincidono
True
>>> f = foo # Le due etichette `f` e `foo` fanno riferimento al medesimo oggetto
>>> f.__name__ # Il nome dell'oggetto a cui `f` fa riferimento è 'foo'
'foo'
>>> 'f' in dir() # Il nome dell'etichetta `f` è `f`
True
```

I nomi delle etichette hanno vari contesti di utilizzo. Ad esempio, essi consentono di recuperare gli oggetti ai quali le etichette fanno riferimento, utilizzando i nomi come chiavi di particolari dizionari che rappresentano i *namespace*. Ne parleremo approfonditamente nel [Capitolo 4](#), ma qui forniremo un breve esempio, in modo da mostrare un caso pratico di utilizzo. A tale scopo consideriamo la funzione built-in `globals()`, la quale restituisce il *namespace globale* sotto forma di dizionario. Quest'ultimo ha per chiavi i nomi delle etichette globali, e per valori i corrispondenti oggetti:

```
>>> a = 44
>>> def foo(x):
...     return x * x
...
>>> d = globals() # Dizionario che ha come chiavi i nomi delle etichette globali
>>> d['a'] # Utilizzo il nome dell'etichetta come chiave, e ottengo come valore l'oggetto
44
>>> d['foo'](10)
100
```

Detto ciò, quando non vi sono possibilità di equivoco parleremo dell'etichetta intendendo però l'oggetto. Ad esempio, nel seguente caso:

```
>>> a, b = 10, 20
>>> a + b
```

piuttosto che dire "abbiamo sommato gli oggetti a cui a e b fanno riferimento", diremo semplicemente che abbiamo sommato a e b . La stessa cosa vale per gli attributi, per cui nel seguente esempio diremo che "abbiamo aggiunto un elemento all'attributo a di Foo ", piuttosto che "abbiamo aggiunto un elemento all'oggetto a cui l'attributo a di Foo fa riferimento":

```
>>> class Foo:
...     a = [1, 2, 3]
...
>>> Foo.a.append(4)
>>> Foo.a
[1, 2, 3, 4]
```

In tutti i casi che possono dare luogo a malintesi saremo invece più formali.

Tipologie di errori

Gli errori in un programma Python si manifestano sotto forma di *errori di sintassi* oppure di *eccezioni*.

Errori di sintassi

Come si può intuire, gli errori di sintassi si verificano quando un'istruzione non viene scritta nel modo corretto. Ad esempio, la mancanza dei due punti al termine dell'istruzione `for` è un errore di sintassi:

```
>>> for i in range(10) print(i) # Mancano i due punti al termine dell'istruzione `for`
      File "<stdin>,, line 1
for i in range(10) print(i)
^
SyntaxError: invalid syntax
```

Gli errori di sintassi vengono segnalati con un messaggio che riporta la linea di codice in cui si è verificato l'errore, e una piccola freccia che indica l'elemento successivo al punto in cui l'errore è stato rilevato. Nel precedente esempio l'errore è stato rilevato prima di `print(i)`, e infatti subito dopo `range(10)` mancano i due punti necessari per terminare l'istruzione `for`. Vengono mostrati anche il nome del file e il numero di linea, in modo da facilitare l'individuazione dell'errore:

```
$ cat foo.py
for i in range(10) print(i) # Mancano i due punti al termine dell'istruzione `for`
$ python foo.py
File "foo.py", line 1
for i in range(10) print(i) # Mancano i due punti al termine dell'istruzione `for`
^
SyntaxError: invalid syntax
```

Gli errori di sintassi vengono segnalati prima che il programma vada in esecuzione, poiché rilevati durante la fase di compilazione in bytecode:

```
$ echo "print(__name__)" > foo.py # Creo uno script contenente una `print()`
$ python foo.py # Quando lo script viene eseguito stampa il nome del modulo
__main__
$ echo "for i in range(10) print(i)" >> foo.py # Aggiungo un errore di sintassi
$ cat foo.py
print(__name__)
for i in range(10) print(i)
```

```
$ python foo.py # Non viene eseguita neppure la prima istruzione
File "foo.py", line 2
for i in range(10) print(i)
^
SyntaxError: invalid syntax
```

Le eccezioni

Le eccezioni vengono sollevate quando un'istruzione, nonostante sia sintatticamente corretta, dà luogo a degli errori durante la sua esecuzione. Si consideri, ad esempio, il file *errors.py*:

```
$ cat errors.py
while True:
x = input('Digita qualcosa (esci con Ctl-D su Unix, Ctl-Z+Return su Win): ')
if x == 'a':
prin('Hai digitato', x) # L'istruzione è sintatticamente corretta
else:
print('Hai digitato', x)
```

È presente un errore di battitura (è stato scritto `prin` anziché `print`), che verrà rilevato solo quando il flusso di esecuzione arriva a quella istruzione, poiché solo in quel momento Python cercherà la definizione dell'etichetta `prin`. Quindi l'errore verrà segnalato solo se l'utente digiterà il carattere `a`:

```
$ python errors.py
Digita qualcosa (esci con Ctl-D su Unix, Ctl-Z+Return su Win): d
Hai digitato d
Digita qualcosa (esci con Ctl-D su Unix, Ctl-Z+Return su Win): u
Hai digitato u
Digita qualcosa (esci con Ctl-D su Unix, Ctl-Z+Return su Win): a
Traceback (most recent call last):
File "errors.py", line 4, in <module>
prin('Hai digitato', x) # L'istruzione è sintatticamente corretta
NameError: name 'prin' is not defined
```

Come si può vedere da questo esempio, quando le eccezioni non vengono gestite dal programma, l'esecuzione termina e viene mostrato un messaggio di errore. Tale messaggio si differenzia da quelli dovuti agli errori di sintassi principalmente per la presenza del *traceback*.

NOTA

Le eccezioni possono essere sollevate anche di proposito, utilizzando l'istruzione `raise`. Consideriamo, ad esempio, il seguente file:


```
$ cat myerror.py
raise Exception('Attenzione, errore in arrivo!')
```

Ecco cosa accade quando lo eseguiamo:

```
$ python myerror.py
Traceback (most recent call last):
File "myerror.py", line 1, in <module>
raise Exception('Attenzione, errore in arrivo!')
Exception: Attenzione, errore in arrivo!
```

Parleremo in modo approfondito dell'istruzione `raise` nel [Capitolo 5](#).

Poiché le eccezioni vengono sollevate solo quando la porzione di codice contenente l'errore viene eseguita, è possibile che un programma funzioni *quasi* sempre. Se avessimo eseguito il file *errors.py* portando avanti 100 cicli, senza mai digitare il carattere `a`, probabilmente ci saremmo sentiti sicuri di poter affermare che il programma era privo di errori. Nel caso si incorra in questo genere di problemi, potrebbe esserci utile usare dei tool che analizzano il codice Python al fine di rilevare eventuali errori. Tra questi citiamo *pyflakes* e *pylint*. Ecco un esempio di utilizzo di quest'ultimo:

```
$ pylint -E errors.py
No config file found, using default configuration
***** Module errors
E: 4: Undefined variable 'prin'
```

La gestione delle eccezioni

In questa sezione introduciamo il meccanismo di gestione delle eccezioni. Consideriamo, a tale scopo, il file *myfile.py*:

```
$ cat myfile.py
mylist = ['a', 'b', 'c', 'd', 'e', 'f']
while True:
data = input('Digita un numero intero compreso tra -%d e %d: '
%(len(mylist), len(mylist) - 1))
num = int(data) # Converto la stringa in numero intero
print(mylist[num])
```

Se lo eseguiamo e digitiamo una stringa che non può essere convertita in intero, la funzione built-in `int()` *solleva* un'eccezione di tipo `ValueError`:

```
$ python myfile.py
Digita un numero intero compreso tra -6 e 5: 2
```

```

c
Digita un numero intero compreso tra -6 e 5: -6
a
Digita un numero intero compreso tra -6 e 5: 2 + 1
Traceback (most recent call last):
File "myfile.py", line 4, in <module>
num = int(data) # Converto la stringa in numero intero
ValueError: invalid literal for int() with base 10: '2 + 1'

```

Possiamo gestire l'eccezione inserendo tra le parole chiave `try` ed `except` la linea logica che dà luogo all'errore. Questo andrà gestito nel blocco di codice che segue la parola chiave `except`:

```

$ cat myfile.py
mylist = ['a', 'b', 'c', 'd', 'e', 'f']
while True:
try:
data = input('Digita un numero intero compreso tra -%d e %d: '
%(len(mylist), len(mylist) - 1))
num = int(data) # Converto la stringa in numero intero
print(mylist[num])
except ValueError:
print('Numero non valido! Inserisci un intero, come 2, 1 e 3')
print('Istruzione successiva alla while')

```

```

$ python myfile.py
Digita un numero intero compreso tra -6 e 5: 4
e
Digita un numero intero compreso tra -6 e 5: 5
f
Digita un numero intero compreso tra -6 e 5: 1 + 2
Numero non valido! Inserisci un intero, come 2, 1 e 3
Digita un numero intero compreso tra -6 e 5: 0
a

```

La parola chiave `try` è una istruzione composta ed `except` è una sua clausola. Quando vengono eseguite le istruzioni della suite del `try`, se non si manifesta alcun errore, allora dall'ultima istruzione del blocco `try` l'esecuzione passa direttamente all'istruzione successiva alla `try/except`, saltando quindi la suite della `except`. Se, invece, una linea logica nella suite del `try` solleva un'eccezione, l'esecuzione da quella linea passa direttamente alla clausola `except`. Questa verifica che il tipo dell'eccezione sollevata corrisponda a quello che si intende gestire, nel qual caso viene eseguita la sua suite in modo da gestire l'errore, in caso contrario la suite `except` viene saltata.

Nel nostro caso la `except` gestisce solo le eccezioni di tipo `ValueError`. Se, quindi, digitiamo un numero che genera un `IndexError` quando si tenta di indicizzare la lista, questa eccezione non verrà gestita:

```
$ python myfile.py
Digita un numero intero compreso tra -6 e 5: 1
b
Digita un numero intero compreso tra -6 e 5: 7
Traceback (most recent call last):
File "myfile.py", line 6, in <module>
print(mylist[num])
IndexError: list index out of range
```

Se si vogliono gestire entrambi i tipi di eccezione nella medesima clausola `except`, si possono inserire i tipi `ValueError` e `IndexError` in una tupla, nel modo seguente:

```
$ cat myfile.py

mylist = ['a', 'b', 'c', 'd', 'e', 'f']
while True:
    try:
        data = input('Digita un numero intero compreso tra -%d e %d: '
                    '%(len(mylist), len(mylist) - 1))
        num = int(data) # Converto la stringa in numero intero
        print(mylist[num])
    except (ValueError, IndexError):
        print('Numero non valido! Inserisci un intero compreso tra -%d e %d'
              '%(len(mylist), len(mylist) - 1))
        print('Istruzione successiva alla while')
```

```
$ python myfile.py
Digita un numero intero compreso tra -6 e 5: 2
c
Digita un numero intero compreso tra -6 e 5: 2 * 2
Numero non valido! Inserisci un intero compreso tra -6 e 5
Digita un numero intero compreso tra -6 e 5: 9
Numero non valido! Inserisci un intero compreso tra -6 e 5
Digita un numero intero compreso tra -6 e 5: 4
e
```

È anche possibile separare la gestione dei differenti tipi di eccezione in diverse clausole `except`:

```
$ cat myfile.py
mylist = ['a', 'b', 'c', 'd', 'e', 'f']
while True:
    try:
        data = input('Digita un numero intero compreso tra -%d e %d: '
                    '%(len(mylist), len(mylist) - 1))
        num = int(data) # Converto la stringa in numero intero
        print(mylist[num])
    except (ValueError, IndexError):
        print('Numero non valido! Inserisci un intero compreso tra -%d e %d'
              '%(len(mylist), len(mylist) - 1))
    except KeyboardInterrupt:
        print('\nHai digitato un carattere di interruzione (Ctl-C), ' +
```

```
'quindi termino il ciclo while')
break
except EOFError:
print('\nHai digitato Ctr-D o Ctr-Z, ma ho deciso di non far ' +
'terminare il ciclo while...')
print('Istruzione successiva alla while')
```

```
$ python myfile.py
Digita un numero intero compreso tra -6 e 5: 3
d
Digita un numero intero compreso tra -6 e 5:
Hai digitato Ctr-D o Ctr-Z, ma ho deciso di non far terminare il ciclo while...
Digita un numero intero compreso tra -6 e 5: 0
a
Digita un numero intero compreso tra -6 e 5: ^C
Hai digitato un carattere di interruzione (Ctl-C), quindi termino il ciclo while
Istruzione successiva alla while
```

Concludiamo questa sezione dicendo che, se nella clausola `except` non viene specificato un tipo di eccezione, verranno catturate le eccezioni di tutti i tipi:

```
$ cat myfile.py
mylist = ['a', 'b', 'c', 'd', 'e', 'f']
while True:
try:
data = input('Digita un numero intero compreso tra -%d e %d: '
'%(len(mylist), len(mylist) - 1))
num = int(data) # Converto la stringa in numero intero
print(mylist[num])
except:
print('Numero non valido! Inserisci un intero compreso tra -%d e %d'
'%(len(mylist), len(mylist) - 1))
print('Istruzione successiva alla while')
```

```
$ python myfile.py
Digita un numero intero compreso tra -6 e 5: -4
c
Digita un numero intero compreso tra -6 e 5: 6
Numero non valido! Inserisci un intero compreso tra -6 e 5
Digita un numero intero compreso tra -6 e 5: 2 * 2
Numero non valido! Inserisci un intero compreso tra -6 e 5
```

Non dobbiamo utilizzare questa modalità solo per pigrizia, perché, se ciò che fa il nostro codice non ci è perfettamente chiaro, ci sono buone probabilità che perdiamo del tempo piuttosto che guadagnarne. Questo perché nella `except` non viene stampato alcun messaggio di errore, per cui, se si verificassero degli errori che non abbiamo previsto, non riusciremmo a capire i motivi del malfunzionamento. Inoltre, questi errori imprevisti verrebbero gestiti alla stessa maniera di quelli che prevedevamo di gestire nella `except`, mentre magari meriterebbero una gestione separata. Questo è il caso migliore, perché, per la

legge di Murphy, la nostra pigrizia, nella maggior parte dei casi, sarà causa di errori logici difficili da rilevare e localizzare.

Nel Capitolo 5 parleremo in dettaglio delle eccezioni e, tra le altre cose, vedremo come il meccanismo dell'ereditarietà ci consenta di catturare in modo appropriato tutte le eccezioni, lasciando propagare quelle che non rappresentano degli errori.

Oggetti iterabili, iteratori e contesto di iterazione

Gli argomenti trattati in questa sezione potrebbero non essere di immediata comprensione, per cui, se incontreremo delle difficoltà, non disperiamo: potremo tornarci sopra in un secondo momento, quando avremo un po' più di dimestichezza con il linguaggio.

Il protocollo di iterazione

Il *protocollo di iterazione* è descritto nella PEP-0234. Definisce il comportamento che deve avere un oggetto contenitore affinché si possa iterare su di esso, ad esempio in un ciclo `for`, in modo da ottenere i suoi elementi uno per volta. Alla base del protocollo sta un oggetto chiamato *iteratore*, avente i seguenti due metodi:

- `iterator.__iter__()`: è un metodo dell'oggetto che restituisce un riferimento all'oggetto stesso;
- `iterator.__next__()`: è un metodo che restituisce l'elemento successivo del contenitore, oppure solleva un'eccezione di tipo `StopIteration` se gli elementi del contenitore sono stati restituiti tutti.

Un oggetto contenitore `obj` è detto *iterabile* se è possibile accedere ai suoi elementi mediante indicizzazione, oppure se è provvisto di un metodo `obj.__iter__()` che restituisce un iteratore.

Sulla base di queste definizioni, un iteratore è un oggetto iterabile, mentre un oggetto iterabile non è detto che sia un iteratore. Ad esempio, un `set` non è un iteratore perché non ha il metodo `set.__next__()`:

```
>>> s = {'a', 3, 'python'}
>>> hasattr(s, '__next__') # I set non sono iteratori
False
```

Vediamo se è un oggetto iterabile. Come prima cosa, verifichiamo che abbia un metodo `set.__iter__()`:

```
>>> hasattr(s, '__iter__')
True
```

Ora dobbiamo verificare che questo metodo restituisca un iteratore. Se così

fosse, il set sarebbe un oggetto iterabile. Procediamo:

```
>>> obj = s.__iter__()
>>> hasattr(obj, '__iter__') # L'oggetto restituito da obj ha il metodo __iter__()
True
>>> obj is obj.__iter__() # Questo metodo di `obj` restituisce `obj` stesso
True
>>> hasattr(obj, '__next__') # L'oggetto `obj` ha anche il metodo `obj.__next__()`
True
```

Manca ancora una condizione per poter affermare che `obj` è un iteratore. Il suo metodo `obj.__next__()` a ogni chiamata deve restituire l'elemento successivo del contenitore e sollevare un'eccezione di tipo `StopIteration` quando gli elementi sono terminati:

```
>>> obj.__next__()
'python'
>>> obj.__next__()
3
>>> obj.__next__()
'a'
>>> obj.__next__()
Traceback (most recent call last):
...
StopIteration
```

Quindi, in definitiva, le istanze del tipo `set` sono oggetti iterabili perché hanno un metodo `set.__iter__()` che restituisce un iteratore, ma non sono iteratori perché non hanno il metodo `set.__next__()`.

Anche le funzioni built-in `iter()` e `next()` consentono di iterare manualmente su un oggetto iterabile:

```
>>> mylist = [1, 2]
>>> iterator = iter(mylist)
>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
Traceback (most recent call last):
...
StopIteration
```

Una caratteristica importante degli iteratori è che restano legati all'oggetto iterabile:

```
>>> mylist = [1, 2]
>>> iterator = iter(mylist)
```

```
>>> next(iterator)
1
>>> next(iterator)
2
>>> mylist.append(3)
>>> next(iterator)
3
```

Approfondiremo questo argomento nel Capitolo 6.

Le classi built-in range ed enumerate

Discuteremo nel Capitolo 2 delle funzioni built-in legate agli oggetti iterabili; per il momento ci limitiamo a parlare delle sole classi `range` ed `enumerate`. La classe `range` ha un argomento `stop` obbligatorio e due argomenti `start` e `step` opzionali:

```
range([start,] stop[, step]) -> range object
```

Restituisce un oggetto iterabile contenente una sequenza di numeri progressivi. Quando è usata con il solo argomento `stop`, allora l'oggetto contiene i numeri da 0 a `stop` escluso:

```
>>> r = range(6)
>>> list(r)
[0, 1, 2, 3, 4, 5]
>>> type(r)
<class 'range'>
>>> hasattr(r, '__iter__'), hasattr(r, '__next__')
(True, False)
>>> i = iter(r)
>>> next(i), next(i), next(i), next(i), next(i)
(0, 1, 2, 3, 4, 5)
>>> next(i)
Traceback (most recent call last):
...
StopIteration
```

Gli argomenti opzionali `start` e `step` consentono di indicare il numero di partenza e l'intervallo tra un numero e il successivo:

```
>>> r = range(3, 19, 2)
>>> list(r)
[3, 5, 7, 9, 11, 13, 15, 17]
>>> r.start, r.stop, r.step
(3, 19, 2)
>>> r = range(-3, -19, -2)
>>> tuple(r)
(-3, -5, -7, -9, -11, -13, -15, -17)
>>> r.count(-4), r.count(-11)
(0, 1)
```



```
(0, 1)
>>> range(3, 10).index(5)
2
```

La classe `enumerate` prende come argomento un oggetto iterabile e restituisce un iteratore che itera su delle tuple. Ogni tupla è composta da un indice progressivo e da un elemento dell'oggetto iterabile. Quando l'oggetto iterabile è una sequenza `seq`, allora l'indice progressivo corrisponde all'indice dell'elemento nella sequenza, per cui le tuple saranno `(idx, seq[idx])`:

```
>>> e = enumerate([1, 2, 3, 4, 5, (), dict()])
>>> type(e)
<class 'enumerate'>
>>> hasattr(e, '__iter__'), hasattr(e, '__next__')
(True, True)
>>> for i in e:
...     print(i, end=' ')
...
(0, 1) (1, 2) (2, 3) (3, 4) (4, 5) (5, ()) (6, {})
>>> for i, j in enumerate(dict(a=[], b=(), c={})):
...     print(i, j)
...
0 c
1 b
2 a
```

Concludiamo questa parte evidenziando una importante differenza tra iteratori e oggetti iterabili che non siano iteratori. Mentre per un oggetto iterabile che non sia un iteratore si possono usare diversi iteratori che iterano su di esso in modo indipendente, questo non è possibile per un oggetto iteratore, poiché il metodo `iteratore.__iter__()` restituisce un riferimento a se stesso:

```
>>> r = range(10) # Le istanze della classe range non sono iteratori
>>> i1 = iter(r)
>>> i2 = iter(r)
>>> next(i1), next(i2) # I due iteratori iterano in modo indipendente
(0, 0)
>>> i1 is i2 # Sono due oggetti distinti
False
>>> e = enumerate([1, 2, 3]) # Le istanze di enumerate sono iteratori
>>> i1 = iter(e)
>>> i2 = iter(e)
>>> next(i1), next(i2) # I due iteratori sono dipendenti
((0, 1), (1, 2))
>>> i1 is i2 is e # Infatti 'i1' e 'i2' fanno riferimento al medesimo oggetto
True
```

Il contesto di iterazione

L'istruzione `for` permette di iterare in modo automatico sugli oggetti iterabili:

```
>>> for i in ('a', 'b', 'c'):
...     print(i)
...
a
b
c
```

Utilizza, infatti, il protocollo di iterazione per compiere in modo automatico la procedura che abbiamo eseguito manualmente nella precedente sezione:

1. ottiene l'iteratore: `iteratore = iter(('a', 'b', 'c'))`;
2. richiede il prossimo elemento chiamando il metodo `iteratore.__next__()`;
3. se `iteratore.__next__()` solleva una eccezione `StopIteration`, allora cattura l'eccezione e fa terminare l'iterazione, altrimenti passa al punto 4;
4. assegna all'etichetta `i` l'elemento successivo: `i = iteratore.__next__()`;
5. esegue il blocco di istruzioni, che in questo caso sono la sola chiamata a `print(i)`;
6. ritorna al punto 2.

Anche i dizionari e i file sono oggetti iterabili. Quando si itera su un dizionario, si itera sulle sue chiavi:

```
>>> for key in {'a': 1, 'b': 2}:
...     print(key) # Il dizionario non è un oggetto ordinato
...
b
a
```

mentre quando si itera su un file, si itera sulle sue linee:

```
>>> open('myfile').read()
'prima linea\nseconda linea\n'
>>> for line in open('myfile'):
...     print(line, end='')
...
prima linea
seconda linea
```

Una importante caratteristica degli oggetti iterabili è che possono essere spaccettati:

```
>>> print(open('myfile').read()) # Il file `myfile` contiene due linee
prima linea
```

seconda linea

```
>>> a, b = open('myfile') # Un file può essere spaccettato
>>> a
'prima linea\n'
>>> b
'seconda linea\n'
>>> a, b = {'nome': 'Emanuela', 'anni': 30}
>>> a, b # Il dizionario non è un oggetto ordinato...
('anni', 'nome')
```

Esercizio conclusivo

In questa sezione faremo un esercizio di riepilogo, che ci consentirà sia di ripassare quanto visto finora, sia di introdurre nuovi concetti. A tale scopo, analizziamo uno script che legge dei file aventi estensione *.data*, effettua alcuni calcoli e poi salva i risultati su altri file. Lo script in questione è il file *dataout.py*:

```
#!/usr/bin/env python
"""Calcolo del valore minimo, massimo e medio dei dati presenti nei file con
estensione '.data'.
```

Legge i file con estensione *.data* presenti nella directory corrente, e di ogni linea calcola il valore massimo, quello minimo e quello medio degli elementi. Salva questi risultati su dei file di output che hanno lo stesso nome dei file di input, ma con estensione *.dataout*.

Lo script prende un argomento opzionale *out_dir_name*, che indica la directory nella quale salvare i file di output. Se questo argomento viene omesso, i file di output vengono salvati in una directory di nome *out*:

```
$ python dataout.py [out_dir_name=out]
```

Se la directory di output non esiste, viene creata.

```
"""
import sys
import os

out_dir_name = sys.argv[1] if sys.argv[1:] else 'out' # Nome directory output
try:
    os.mkdir(out_dir_name) # Crea la directory di output
    print('I file verranno scritti nella directory', out_dir_name)
except FileExistsError:
    print("I file verranno scritti nella directory", out_dir_name, "esistente")

for file_name in os.listdir():
    if file_name.endswith('.data'):
        out_file_name = os.path.join(out_dir_name, file_name + 'out')
        print('Sto per scrivere sul file `%s` ...' % out_file_name, end=' ')
        out = open(out_file_name, 'w')
        for line in open(file_name):
            d = [float(item) for item in line.split()] # Dati della linea
            out.write('%0.2f %0.2f %0.2f\n' % (min(d), max(d), sum(d)/len(d)))
        print('Fatto!')
```

Il codice sorgente di tutti gli esercizi conclusivi e degli esempi più significativi del libro è disponibile all'URL <http://code.google.com/p/the-pythonic-way/>.

Probabilmente buona parte di questo codice ci è oscuro, ma non preoccupiamoci, lo analizzeremo riga per riga nelle sezioni che seguono. Per il momento leggiamolo ancora e sforziamoci il più possibile di capirne il significato in modo autonomo.

Esecuzione dello script

Lo script legge dalla directory corrente i file con estensione *.data* e per ognuno di essi crea un file di output con lo stesso nome, ma con estensione *.dataout*. Ad esempio, se trova un file di input chiamato *20121218.data*, lo apre, lo elabora e poi salva i risultati in un file di output chiamato *20121218.dataout*.

Lo script scrive su ogni linea dei file di output tre valori: il valore minimo, quello massimo e quello medio delle corrispondenti linee dei file di input. Ad esempio, se lo script trova nella directory corrente il seguente file di input:

```
$ cat 20100309.data # File avente una sola linea
10.0090.0005.0039.5055.50
```

lo apre, calcola il valore minimo, quello massimo e quello medio degli elementi della linea e li scrive nel file di output:

```
$ cat out/20100309.dataout
5.0090.0040.00
```

Se il file di input ha più di una linea, come, ad esempio, il seguente:

```
$ cat 20110315.data
696.13437.85005.92332.10259.26
883.39007.15476.95582.44259.92
004.96134.77695.30257.56763.61
679.62265.94326.11366.53009.39
```

il file di output avrà altrettante linee, ciascuna delle quali conterrà il valore minimo, quello massimo e quello medio della corrispondente linea del file di input:

```
$ cat out/20110315.dataout
```

```
5.92696.13346.25
7.15883.39441.97
4.96763.61371.24
9.39679.62329.52
```

Lo script accetta un argomento opzionale da linea di comando, che rappresenta il nome della directory nella quale salvare i file. Se non gli viene passato questo argomento, allora i file di output verranno salvati in una directory chiamata *out*:

```
$ ls # Contenuto della directory di lavoro prima dell'esecuzione dello script
20100309.data 20110315.data 20121218.data dataout.py data.txt errors.py
$ python dataout.py # Non passiamo allo script il nome della directory di output
I file verranno scritti nella directory `out`
Sto per scrivere sul file `out/20110315.dataout` ... Fatto!
Sto per scrivere sul file `out/20100309.dataout` ... Fatto!
Sto per scrivere sul file `out/20121218.dataout` ... Fatto!
$ ls # Lo script ha creato la directory `out`
20100309.data 20110315.data 20121218.data dataout.py data.txt errors.py out
$ ls out # I file di output sono stati salvati nella directory `out`
20100309.dataout 20110315.dataout 20121218.dataout
```

Se, invece, passiamo da riga di comando un argomento, questo verrà utilizzato come nome della directory di output:

```
$ ./dataout.py pippo # Verrà creata la directory `pippo`, nella quale salvare i file
I file verranno scritti nella directory `pippo`
Sto per scrivere sul file `pippo/20110315.dataout` ... Fatto!
Sto per scrivere sul file `pippo/20100309.dataout` ... Fatto!
Sto per scrivere sul file `pippo/20121218.dataout` ... Fatto!
$ ls
20100309.data 20110315.data 20121218.data dataout.py data.txt errors.py out pippo
$ ls pippo
20100309.dataout 20110315.dataout 20121218.dataout
```

Se la directory di output esiste, lo script lo segnala con il primo messaggio di stampa e poi continua la sua esecuzione:

```
$ ./dataout.py
I file verranno scritti nella directory out esistente
Sto per scrivere sul file `out/20110315.dataout` ... Fatto!
Sto per scrivere sul file `out/20100309.dataout` ... Fatto!
Sto per scrivere sul file `out/20121218.dataout` ... Fatto!
```

Ora che sappiamo cosa fa lo script, vediamo di capire il significato del suo codice. Il primo passo consiste nello scoprire come lo script legge gli argomenti passati da linea di comando.

Passaggio degli argomenti da linea di comando

Gli argomenti passati al programma da linea di comando vengono memorizzati da Python in una lista accessibile tramite il modulo `sys`. Questa lista si chiama `sys.argv` e contiene, come primo elemento, il nome del file, e come restanti elementi gli altri argomenti passati da linea di comando:

```
$ cat foo.py # Lo script `foo.py` stampa il contenuto della lista `sys.argv`
import sys
print(sys.argv)
$ python foo.py # In questo caso sys.argv contiene solo il nome del file
['foo.py']
$ python foo.py aaa # Passiamo a `python`, oltre al nome del file, anche un altro argomento
['foo.py', 'aaa']
$ python foo.py aaa -h bbb -xyz
['foo.py', 'aaa', '-h', 'bbb', '-xyz']
```

NOTA

Il nome *argv* sta per *argument vector* e deriva dal C, dove viene usualmente utilizzato per indicare il parametro al quale assegnare gli argomenti passati da linea di comando:

```
int main(int argc, char *argv[]) {} // Codice C
```

Gli elementi di `sys.argv` sono delle stringhe, per cui è necessario convertirli nel tipo corretto per poterli utilizzare in modo appropriato:

```
$ cat foo.py
import sys
a = int(sys.argv[1]) # Converto l'argomento in un numero intero
print(a ** 2)
$ python foo.py 2
4
$ python foo.py 4
16
```

Nel nostro script il nome della directory di output viene assegnato con l'istruzione:

```
out_dir_name = sys.argv[1] if sys.argv[1:] else 'out' # Nome directory di output
```

L'espressione condizionale alla destra del simbolo di uguale valuta come espressione di test uno slicing di `sys.argv`, precisamente `sys.argv[1:]`. Come abbiamo già detto, data una lista `mylist`, lo slicing `mylist[i:j]` restituisce una lista degli elementi

di `mylist` a partire da quello di indice `i` sino a quello di indice `j` escluso. Se l'indice `j` viene omesso, allora lo slicing restituisce tutti gli elementi a partire da quello con indice `i` sino al termine della lista. Consideriamo, ad esempio, il seguente script:

```
$ cat slicing_example.py
import sys
print('sys.argv:', sys.argv, sep=' ')
print('sys.argv[0:2]', sys.argv[0:2], sep=' ') # Dallo 0 al 2 (escluso)
print('sys.argv[0:]', sys.argv[0:], sep=' ') # Tutti gli elementi
print('sys.argv[1:]', sys.argv[1:], sep=' ') # Da quello con indice 1 in poi
```

ed eseguiamolo due volte. La prima volta non gli passiamo alcun argomento da linea di comando, mentre la seconda volta gli passiamo tre argomenti:

```
$ python slicing_example.py
sys.argv: ['slicing_example.py']
sys.argv[0:2] ['slicing_example.py']
sys.argv[0:] ['slicing_example.py']
sys.argv[1:] []
$ python slicing_example.py aaa bbb ccc
sys.argv: ['slicing_example.py', 'aaa', 'bbb', 'ccc']
sys.argv[0:2] ['slicing_example.py', 'aaa']
sys.argv[0:] ['slicing_example.py', 'aaa', 'bbb', 'ccc']
sys.argv[1:] ['aaa', 'bbb', 'ccc']
```

Tornando al nostro codice, quando passiamo a `python` solo il nome del file, allora la lista `sys.argv[1:]` risulta vuota. Poiché una lista vuota viene valutata come `False` in un test di verità:

```
>>> out_dir_name = 'lista non vuota' if [] else 'lista vuota'
>>> out_dir_name
'lista vuota'
```

l'espressione condizionale `sys.argv[1] if sys.argv[1:] else 'out'` restituisce `'out'` e quindi, in definitiva, si ha l'assegnamento `out_dir_name = 'out'`.

Se, invece, viene passato un ulteriore argomento oltre al nome del file, allora `sys.argv[1:]` non risulta vuota e viene valutata come `True`. In questo caso, quindi, si ha l'assegnamento `out_dir_name = sys.argv[1]`.

Se volessimo effettuare il parsing degli argomenti passati da linea di comando, in modo che vengano mostrati un `help` e un messaggio di utilizzo, o che vengano gestiti gli errori nel caso in cui siano passati al programma degli argomenti non validi, dovremmo utilizzare il modulo `argparse` della libreria standard. Questo effettua il parsing degli elementi di `sys.argv`, genera

automaticamente i messaggi di utilizzo e gestisce anche gli errori. Vedremo un esempio di utilizzo di questo modulo nel Capitolo 2.

L'interfaccia con il sistema operativo

Il nostro script deve potersi interfacciare con il sistema operativo, sia per creare la directory di output, nel caso non esista, sia per cercare i file con estensione *.data* all'interno della directory di lavoro. Come abbiamo già accennato in precedenza, è il modulo `os` a offrire questa interfaccia. Infatti nello script abbiamo usato la funzione `os.mkdir()` per creare le directory di output:

```
import os
os.mkdir(out_dir_name) # Crea la directory di output
```

Se la directory esiste, `os.mkdir()` solleva una eccezione di tipo `FileExistsError`:

```
$ ls # La directory di lavoro è vuota
$ python -c "import os; os.mkdir('foodir')"
$ ls
foodir
$ python -c "import os; os.mkdir('foodir')"
Traceback (most recent call last):
...
FileExistsError: [Errno 17] File exists: 'foodir'
```

Per questo motivo l'istruzione che contiene `os.mkdir()` è stata inserita all'interno di una istruzione `try`. In questo modo, se la directory esiste, viene sollevata un'eccezione e il flusso di esecuzione passa direttamente alla clausola `except`, saltando la `print()` subito sotto `os.mkdir()`. La suite della `except`, a questo punto, stampa un messaggio, avvisando che la directory esiste, e l'esecuzione passa poi all'istruzione `for`.

La funzione `os.listdir()` chiamata senza argomenti restituisce una lista dei nomi di file e directory presenti all'interno della directory corrente:

```
>>> import os
>>> os.listdir()
['dataout.py', '20110315.data', '20100309.data', 'errors.py', 'data.txt', '20121218.data']
```

Il ciclo `for`, quindi, itera sulla lista restituita da `os.listdir()` ed esegue delle azioni solo quando il nome del file termina con *.data*. I file con estensione *.data* sono i seguenti:

```
>>> for file_name in os.listdir():
```

```
... if file_name.endswith('.data'):
...     print(file_name)
...
20110315.data
20100309.data
20121218.data
```

Per ognuno di essi deve essere creato un file di output il cui nome è dato da:

```
out_file_name = os.path.join(out_dir_name, file_name + 'out') # Nome del file di output
```

La funzione `os.path.join()` unisce uno o più path, usando come separatore quello adottato dal sistema operativo in uso, quindi uno slash nei sistemi Unix-like:

```
>>> import sys, os
>>> sys.platform # Stiamo lavorando su Linux
'linux'
>>> print(os.sep) # Questo è il separatore usato nei sistemi Unix-like
/
>>> mypath = os.path.join('primo', 'secondo', 'terzo') # Unisce i path
>>> print(mypath)
primo/secondo/terzo
```

e un backslash nei sistemi Windows:

```
>>> import sys, os
>>> sys.platform # Stiamo lavorando su Windows
'win32'
>>> print(os.sep) # Questo è il separatore usato nei sistemi Windows
\
>>> mypath = os.path.join('primo', 'secondo', 'terzo') # Unisce i path
>>> print(mypath)
primo\secondo\terzo
```

Nel nostro caso il path finale è dato dall'unione del nome della directory di output con il nome del file di output. Ad esempio, se il nome della directory di output è `out_dir_name = 'outdir'` e quello del file di input è `file_name = '20110315.data'`, abbiamo:

```
>>> out_dir_name = 'outdir' # Nome della directory di output
>>> file_name = '20110315.data' # Nome del file di input
>>> file_name + 'out' # Nome del file di output
'20110315.dataout'
>>> os.path.join(out_dir_name, file_name + 'out') # Path finale
'outdir/20110315.dataout'
```

Se, anziché usare `os.path.join()`, avessimo unito i percorsi inserendo il separatore

specifico del nostro sistema operativo, lo script non sarebbe stato *portabile*, ossia non avremmo potuto eseguirlo alla stessa maniera su tutti i sistemi operativi. Per intenderci, se avessimo creato il path in questo modo:

```
>>> 'out\myfile.dataout'
```

utilizzando il separatore specifico dei sistemi Windows, il nostro programma non avrebbe funzionato correttamente sui sistemi Unix-like. Consideriamo, ad esempio, il seguente script:

```
$ cat myscript.py
import os
os.mkdir('out')
f = open('out\myfile.dataout', 'w')
```

Quando lo eseguiamo su Windows, questo crea una directory di nome *out*, dopodiché al suo interno crea il file *myfile.dataout*. Il path passato a `open()`, però, utilizza il separatore specifico di Windows, per cui lo script non è portabile. Infatti su un sistema Unix-like non viene creato un file *myfile.dataout* all'interno della directory *out*, ma piuttosto un file *out\myfile.dataout* nella directory corrente:

```
$ ls # La directory corrente contiene solamente lo script
myscript.py
$ python myscript.py # Eseguiamo lo script su Linux, dove il separatore è '/' e non '\'
$ ls out # La directory 'out' è vuota
$ ls # Infatti la 'open()' ha creato un file di nome 'out\myfile.dataout'
myscript.py out out\myfile.dataout
```

È molto importante, quindi, conoscere il modulo `os`, poiché fornisce tanti strumenti che ci consentono di scrivere dei programmi portabili.

NOTA

L'oggetto a cui fa riferimento `os.path` è un modulo, che è differente a seconda del sistema operativo in uso. Ad esempio, nei sistemi Unix-like `os.path` fa riferimento al modulo `posixpath`:

```
>>> import os.path
>>> os.path
<module 'posixpath' from '/usr/local/lib/python3.4/posixpath.py'>
>>> import posixpath
>>> posixpath is os.path
True
```

mentre nei sistemi Windows fa riferimento al modulo `ntpath`:

```
>>> import os.path
>>> os.path
<module 'ntpath' from 'C:\\Python33\\lib\\ntpath.py'>
>>> import ntpath
>>> ntpath is os.path
True
```

Quindi, se chiamiamo la funzione `os.path.join()` da un sistema Unix-like, in realtà la funzione che stiamo chiamando è `posixpath.path.join()`, mentre se chiamiamo `os.path.join()` da un sistema Windows, stiamo in realtà chiamando la funzione `ntpath.path.join()`.

Concludiamo questa introduzione al modulo `os` con un consiglio: leggete la documentazione online sul sito ufficiale, che trovate alla pagina <http://docs.python.org/3/library/os.html>.

List comprehension

Il ciclo `for` più interno itera sulle linee del file. Per ciascuna linea viene creata una lista degli elementi convertiti in `float`:

```
>>> for line in open('20100309.data'):
...     data = [float(item) for item in line.split()]
...     line # Quando lavoriamo in modo interattivo non abbiamo bisogno di usare la `print()`
...     data # La shell interattiva, infatti, mostra la rappresentazione dell'oggetto
...
'10.0090.0005.0039.5055.50\n'
[10.0, 90.0, 5.0, 39.5, 55.5]
```

L'espressione `[float(item) for item in line.split()]` è una list comprehension. Se facciamo mente locale, ricorderemo che ne abbiamo già parlato nella sezione *La funzione built-in `help()` e le stringhe di documentazione*, dove avevamo detto che è una espressione che consente di creare una lista con una sintassi elegante e compatta. Riprendiamo ancora l'argomento, riservandoci di discuterne in dettaglio nel [Capitolo 2](#).

Vediamo di capire cosa fa esattamente la nostra list comprehension, partendo dal metodo `line.split()`:

```
>>> line = '10.0090.0005.0039.5055.50\n'
>>> line.split()
['10.00', '90.00', '05.00', '39.50', '55.50']
```

Come possiamo vedere, `line.split()` restituisce una lista degli elementi della stringa. Questa lista non può, però, essere utilizzata per il calcolo della somma, perché i suoi elementi sono stringhe e non numeri:

```
>>> sum(line.split())
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

È quindi necessario ottenere una lista i cui elementi siano `float` e non `str`. Una soluzione può essere la seguente:

```
>>> line = '10.0090.0005.0039.5055.50\n'
>>> data = []
>>> for item in line.split():
...     data.append(float(item))
...
>>> data
[10.0, 90.0, 5.0, 39.5, 55.5]
```

La soluzione che, invece, abbiamo adottato nel nostro script fa uso di una `list comprehension` e, come possiamo vedere, è molto più compatta ed elegante della precedente:

```
>>> line.split()
['10.00', '90.00', '05.00', '39.50', '55.50']
>>> data = [float(item) for item in line.split()]
>>> data
[10.0, 90.0, 5.0, 39.5, 55.5]
```

La sua sintassi è autoesplicativa: ogni elemento `item` di `line.split()` viene convertito in `float` e inserito nella lista.

Espressioni di formattazione delle stringhe di testo

Ci manca ancora un tassello per completare il puzzle:

```
out.write('%0.2f%0.2f%0.2f\n' %(min(d), max(d), sum(d)/len(d)))
```

NOTA

A partire da Python 3.4, il modo migliore per calcolare la media consiste nell'utilizzare la funzione `mean()` del modulo `statistics`. Per maggiori informazioni si consulti la documentazione ufficiale, alla pagina

Il simbolo % all'interno di una stringa di per sé viene stampato come tutti gli altri caratteri:

```
>>> 'ciao %f'
'ciao %f'
```

Quando, però, la stringa è seguita da un simbolo %, a sua volta seguito da un letterale o da una etichetta, allora il %f all'interno della stringa ha un significato speciale:

```
>>> 'ciao %f' % 11.33
'ciao 11.330000'
>>> num = 11.33
>>> 'ciao %f' % num
'ciao 11.330000'
```

Come possiamo vedere, il %f è stato sostituito rispettivamente dal letterale e dall'etichetta che seguono il simbolo % esterno alla stringa. Il carattere f sta a indicare che il letterale o l'etichetta vanno convertiti in float:

```
>>> 'ciao %f' % 100
'ciao 100.000000'
```

È possibile anche specificare il numero delle cifre decimali da visualizzare. Per fare ciò si fa precedere la f da .numero_cifre, come mostrato di seguito:

```
>>> 'ciao %.2f' % 100
'ciao 100.00'
>>> 'ciao %.3f' % 100
'ciao 100.000'
```

Se vogliamo formattare più oggetti contemporaneamente, dobbiamo racchiuderli in una tupla dopo il simbolo % esterno alla stringa, come mostrato di seguito:

```
>>> a, b, c = 11, 22, 33
>>> '%.2f %.2f %.2f' % (a, b, c)
'11.00 22.00 33.00'
```

Oltre alla formattazione con conversione in float tramite %f, è possibile utilizzare altre codifiche. Ad esempio, %d converte in intero, il %s in stringa, e via

dicendo. Vedremo tutte le possibili modalità di formattazione delle stringhe nel Capitolo 2.

Se rileggiamo lo script *dataout.py*, ci renderemo conto che tutto ciò che inizialmente ci sembrava incomprensibile ha acquistato un senso.

Ora che abbiamo una visione d'insieme del linguaggio, possiamo affrontare con disinvoltura i prossimi capitoli.

Il cuore del linguaggio

Questo capitolo è focalizzato prevalentemente sui tipi del **core data type** e le loro istanze: **numeri**, **stringhe**, **liste**, **tuple**, **dizionari** e **set**. Ci soffermeremo a descrivere nel dettaglio i **meccanismi di codifica** delle stringhe e il legame tra **byte** e **caratteri**. Parleremo ancora di **oggetti iterabili** e, al termine del capitolo, vedremo come effettuare il **parsing** degli argomenti passati da linea di comando, come gestire le **date** e il **tempo** e come lavorare con le **collezioni**.

Numeri

La famiglia dei tipi built-in numerici è la seguente:

- il tipo `int`, le cui istanze rappresentano dei *numeri interi*;
- il tipo `float`, le cui istanze rappresentano i *numeri floating point*;
- il tipo `complex`, le cui istanze rappresentano i *numeri complessi*;
- il tipo `bool`, che ha due sole istanze, alle quali si fa riferimento tramite le parole chiave `True` e `False`.

Le istanze dei tipi numerici, che chiameremo *numeri*, sono oggetti immutabili e risultano istanze anche del tipo `numbers.Number`:

```
>>> import numbers
>>> isinstance(33, int), isinstance(33, numbers.Number)
(True, True)
>>> isinstance(33.3, float), isinstance(33.3, numbers.Number)
(True, True)
>>> isinstance(33.3 + 1j, complex), isinstance(33.3 + 1j, numbers.Number)
(True, True)
>>> isinstance(False, bool), isinstance(False, numbers.Number)
(True, True)
```

Il mondo dei numeri in Python non è legato ai soli tipi numerici built-in e alle loro istanze, ma, come vedremo, coinvolge anche alcuni moduli della libreria standard e diverse funzioni built-in.

Numeri interi

I numeri interi in Python non sono limitati inferiormente o superiormente e ogni *intero* ha quattro rappresentazioni *letterali*:

1. la rappresentazione in base 10, detta *decimale*: è quella di default, e gli interi sono rappresentati mediante sole cifre decimali, come ad esempio i numeri 12 (dodici) e 99 (novantanove);
2. la rappresentazione in base 2, detta *binaria*: gli interi sono rappresentati mediante la loro codifica in binario, preceduta da `0b`; ad esempio, la rappresentazione binaria del numero 7 decimale è `0b111`;
3. la rappresentazione in base 8, detta *ottale*: gli interi sono rappresentati mediante la loro codifica ottale, preceduta da `0o`; ad esempio, la rappresentazione ottale del numero 10 decimale è `0o12`;

4. la rappresentazione in base 16, detta *esadecimale*: gli interi sono rappresentati mediante la loro codifica esadecimale, preceduta da 0x: ad esempio, il numero 10 decimale è dato da 0xA.

Il numero intero -15 può quindi essere rappresentato nei seguenti modi:

```
>>> -15 # Rappresentazione decimale
-15
>>> -0b1111 # Rappresentazione binaria
-15
>>> -0o17 # Rappresentazione ottale
-15
>>> -0xF # Rappresentazione esadecimale
-15
>>> type(-15), type(-0b1111), type(-0o17), type(-0xF)
(<class 'int'>, <class 'int'>, <class 'int'>, <class 'int'>)
```

Le lettere all'interno dei letterali possono essere indicate indifferentemente in maiuscolo o in minuscolo:

```
>>> 0b1010, 0B1010 # Rappresentazione binaria del numero `10`
(10, 10)
>>> 0o12, 0O12 # Rappresentazione ottale del numero `10`
(10, 10)
>>> 0xA, 0xa, 0Xa # Rappresentazione esadecimale del numero `10`
(10, 10, 10)
```

NOTA

Se non è chiaro il significato delle diverse rappresentazioni numeriche, si può consultare la pagina http://it.wikipedia.org/wiki/Sistemi_di_numerazione. Nelle voci correlate sono riportati i riferimenti ai vari sistemi di numerazione, compresi quello decimale, binario, ottale ed esadecimale.

Dato un numero in rappresentazione decimale, le funzioni built-in `bin()`, `oct()` e `hex()` consentono di ottenere, rispettivamente, la sua rappresentazione binaria, ottale ed esadecimale sotto forma di stringa:

```
>>> bin(15) # Restituisce la rappresentazione binaria del numero 15
'0b1111'
>>> oct(15) # Restituisce la rappresentazione ottale del numero 15
'0o17'
>>> hex(15) # Restituisce la rappresentazione esadecimale del numero 15
'0xf'
```

Per convertire il contenuto di una stringa in un numero intero si può utilizzare la classe `int`:

```
>>> int("44") # Per default considera il numero in rappresentazione decimale
44
```

Questa può essere chiamata passandole un secondo argomento opzionale, utilizzato per indicare la base con cui è rappresentato il numero:

```
>>> int("44", 8) # Informiamo `int` che 44 è in rappresentazione ottale
36
```

I numeri per default sono considerati da `int` in rappresentazione decimale, per cui, se le si passa una rappresentazione differente e non si specifica la base, viene sollevata un'eccezione:

```
>>> int("0o44") # La base di default è 10
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: '0o44'
>>> int("0o44", 8) # Quindi in questo caso il secondo argomento è obbligatorio
36
```

Ecco qualche altro esempio:

```
>>> int("44", 16), int("0x44", 16) # Da esadecimale a decimale
(68, 68)
>>> int("100", 2), int("0b100", 2) # Da binario a decimale
(4, 4)
>>> int("200", 2) # Le rappresentazioni binarie possono contenere solo le cifre `0` e `1`...
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 2: '200'
```

Anche la funzione built-in `eval()` converte una stringa in un numero, ma il suo comportamento è più generale, poiché valuta ed esegue delle espressioni contenute in una stringa, restituendone il risultato:

```
>>> eval("-15"), eval("-0b1111"), eval("-0o17"), eval("-0xF")
(-15, -15, -15, -15)
>>> eval("-15 * -0b1111 * -0o17 * -0xF") # Valuta l'espressione e la esegue
50625
```

Operatori che consentono di processare i numeri interi

In questa sezione vedremo le più importanti operazioni che possiamo eseguire

con gli interi.

Elevamento a potenza

Come ormai sappiamo, il simbolo `**` rappresenta l'operatore di elevamento a potenza:

```
>>> 2 ** 4 # Elevamento a potenza: 2 elevato 4
16
```

I numeri interi non hanno limite superiore e inferiore, per cui il seguente elevamento a potenza non genera alcun errore, ma viene calcolato in modo corretto:

```
>>> num = 2 ** 1000000 # Meglio non mandarlo in output...
>>> len(str(num)) # Perché il numero di cifre è... 301030!
301030
```

NOTA

In Python 2 i numeri interi non sono rappresentati dal solo tipo `int`, ma anche dal tipo `long`. Se il valore del numero intero è compreso tra `-sys.maxint - 1` e `sys.maxint` viene rappresentato con il tipo `int`, altrimenti con il tipo `long`:

```
>>> import sys
>>> sys.maxint
2147483647
>>> type(-sys.maxint - 1), type(sys.maxint)
(<type 'int'>, <type 'int'>)
>>> type(-sys.maxint - 2), type(sys.maxint + 1)
(<type 'long'>, <type 'long'>)
```

Il valore di `sys.maxint` dipende dal numero di byte utilizzati per rappresentare le variabili di tipo `int` in linguaggio C, per cui è legato all'architettura della nostra macchina e al sistema operativo in uso. Ad esempio, in una macchina con architettura a 64 bit e sistema operativo Unix-like, `sys.maxint` è pari a 9223372036854775807, mentre in un sistema Windows su architettura a 64 bit è pari a 2147483647.

Divisione vera e floor

L'operatore `/` effettua una divisione tra interi, e genera un `float` (vedi PEP-0238):

```
>>> 6 / 5 # La divisione tra due `int` genera un `float`
```

Viene sempre generato un `float`, anche quando il risultato potrebbe essere un `int`:

```
>>> type(2/1) # Genera sempre un 'float', anche quando il risultato potrebbe essere un 'int'
<class 'float'>
```

L'operatore `//`, indipendentemente dal tipo dei suoi operandi, *arrotonda* il risultato della divisione all'intero inferiore. Questo tipo di divisione è detto *divisione floor*, e genera un `int` nel caso in cui entrambi gli operandi siano interi:

```
>>> 5 // 2, -5 // 2, 4 // 2, 5.0 // 2
(2, -3, 2, 2.0)
```

NOTA

Il significato dell'operatore `/` è uno dei più importanti punti di rottura tra Python 2 e Python 3. In Python 2, infatti, questo operatore effettua una *divisione floor*:

```
>>> -5 / 2 # In Python 2 effettua una divisione floor
-3
```

Possiamo ottenere una divisione vera anche in Python 2:

```
>>> from __future__ import division
>>> -5 / 2
-2.5
```

Operatori di confronto

Il simbolo `=` rappresenta l'*operatore di uguaglianza* e consente di verificare se due oggetti hanno uguale *valore*:

```
>>> x, y = 2 ** 1000, 2 ** 1000 # 'x' e 'y' fanno riferimento ad oggetti di ugual valore
>>> x == y
True
```

Anche se due oggetti hanno uguale valore, non è detto che siano lo stesso oggetto in memoria. Come abbiamo visto nel [Capitolo 1](#), dobbiamo usare la parola chiave `is` per sapere se due oggetti hanno la stessa identità:

```
>>> x is y
False
>>> id(x), id(y) # Le due etichette effettivamente fanno riferimento a oggetti diversi
(3071280048, 3071280200)
```

NOTA

Come abbiamo già detto nel Capitolo 1, quando Python incontra i letterali dei tipi immutabili, per questioni di ottimizzazione (indipendentemente dall'implementazione) predilige il riutilizzo di oggetti già esistenti piuttosto che la creazione di oggetti nuovi, specie se si tratta di oggetti che occupano poco spazio in memoria:

```
>>> x, y = 2 ** 5, 2 ** 5
>>> x == y
True
>>> x is y # In questo caso 'x' e 'y' fanno riferimento al medesimo oggetto
True
```

L'operatore `!=` restituisce `True` se i suoi operandi hanno diverso valore:

```
>>> x = 10
>>> y = 10
>>> x != y # I due oggetti hanno diverso valore?
False
>>> x is y
True
```

Gli operatori `<`, `<=`, `>` e `>=` restituiscono `True` se il loro primo operando è, rispettivamente, *minore*, *minore o uguale*, *maggiore*, *maggiore o uguale* al loro secondo operando:

```
>>> 44 < 55
True
>>> 44 < 44, 44 <= 44
(False, True)
>>> 55 > 33
True
>>> 55 > 55, 55 >= 55.0
(False, True)
```

Questi operatori possono anche essere legati l'uno all'altro:

```
>>> 44 < 55 > 33 # Il 55 è maggiore sia di 44 che di 33?
True
>>> 33 < 44 < 55 # Il 44 è compreso tra il 33 e il 55?
```

Operatori di shift

Gli operatori di *shift a sinistra* e *shift a destra* sono rappresentati, rispettivamente, dai simboli `<<` e `>>`. Per mostrare il loro funzionamento utilizziamo la rappresentazione binaria:

```
>>> 0b1011
11
>>> bin(0b1011 << 3) # Vengono aggiunti 3 zeri come bit meno significativi
'0b1011000'
```

Dato un numero `num`, effettuare il suo shift a sinistra di `n` posizioni equivale a moltiplicarlo per `2 ** n`:

```
>>> 0b1011 << 3, 0b1011 * 2**3 # `num << 3` equivale a `num * 2**3`
(88, 88)
```

Concludiamo con qualche esempio di shift a destra:

```
>>> bin(0b1011 >> 3) # Shift a destra di 3 bit
'0b1'
>>> bin(0b1011 >> 5) # Shift a destra di 5 bit, ma non si va oltre lo zero
'0b0'
```

Le enumerazioni di interi

Con la PEP-0435 (Python 3.4) sono state introdotte le enumerazioni. Queste non fanno parte dei tipi built-in, per cui, se vogliamo utilizzarle, dobbiamo importare il modulo `enum`. In questa sezione ci occuperemo solamente delle enumerazioni di interi, mentre nel [Capitolo 6](#) vedremo in dettaglio le enumerazioni generiche.

Le enumerazioni di interi sono istanze del tipo `enum.IntEnum`:

```
>>> from enum import IntEnum
>>> Pasta = IntEnum('Pasta', 'spaghetti lasagne mezze_penne')
```

Il primo argomento passato a `IntEnum` è il nome che abbiamo voluto dare all'enumerazione:

```
>>> Pasta.__name__
'Pasta'
```

Il secondo argomento è una sequenza contenente i *membri* (gli attributi)

dell'enumerazione. Ad esempio, alla stringa `'spaghetti lasagne mezze_penne'` corrispondono tre membri:

```
>>> Pasta.spaghetti
<Pasta.spaghetti: 1>
>>> Pasta.lasagne
<Pasta.lasagne: 2>
>>> Pasta.mezze_penne
<Pasta.mezze_penne: 3>
```

NOTA

Abbiamo utilizzato l'etichetta `Pasta` (con iniziale maiuscola), perché le istanze di `IntEnum` sono delle classi. Approfondiremo questo concetto nel [Capitolo 6](#), quando parleremo delle metaclassi.

Quando i membri vengono indicati tramite una stringa, come nel caso precedente, oltre che da uno spazio possono essere separati anche da una virgola:

```
>>> Pasta = IntEnum('Pasta', 'spaghetti, lasagne, mezze_penne')
>>> Pasta.spaghetti
<Pasta.spaghetti: 1>
```

In generale, possono essere espressi come elementi di una sequenza:

```
>>> Pasta = IntEnum('Pasta', ('spaghetti', 'lasagne', 'mezze_penne'))
>>> Pasta.spaghetti
<Pasta.spaghetti: 1>
```

I membri hanno un attributo `name`, che è una stringa rappresentativa del nome del membro, e un attributo `value`, che rappresenta il valore:

```
>>> Pasta.spaghetti.name
'spaghetti'
>>> Pasta.spaghetti.value
1
```

Gli oggetti di tipo `enum.IntEnum` sono detti enumerazioni di interi perché i loro membri sono degli interi:

```
>>> isinstance(Pasta.spaghetti, int)
True
```


e si comportano come tali:

```
>>> int(Pasta.spaghetti)
1
>>> int(Pasta.lasagne)
2
>>> Pasta.spaghetti + Pasta.lasagne
3
>>> Pasta.spaghetti == 1
True
```

Possono essere usati ovunque, al posto di un intero:

```
>>> ('a', 'b', 'c')[Pasta.spaghetti]
'b'
```

pertanto possiamo compararli tra loro o con altri numeri reali:

```
>>> Pasta.spaghetti < Pasta.lasagne
True
>>> Pasta.spaghetti < 1.22
True
```

I membri di una enumerazione sono read-only, per cui non possono essere riassegnati:

```
>>> Pasta.spaghetti = 33
Traceback (most recent call last):
...
AttributeError: Cannot reassign members.
```

e neppure cancellati:

```
>>> del Pasta.spaghetti
Traceback (most recent call last):
...
AttributeError: Pasta: cannot delete Enum member.
```

Neppure il loro valore può essere riassegnato:

```
>>> Pasta.spaghetti.value = 33
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

Parleremo in dettaglio delle enumerazioni nell'esercizio conclusivo del Capitolo 6.

Aritmetica dei numeri interi

Abbiamo detto che il tipo `int` consente di rappresentare qualsiasi intero, senza alcuna eccezione. Solo la memoria della nostra macchina può limitare il range di valori ammissibili, visto che il numero di byte necessari per codificare un intero aumenta con l'aumentare del valore assoluto dell'intero stesso. Possiamo verificare quanto appena detto utilizzando la funzione `sys.getsizeof()`, la quale restituisce la dimensione di un oggetto espressa in byte:

```
>>> import sys
>>> sys.getsizeof(20)
28
>>> sys.getsizeof(20**100)
84
>>> sys.getsizeof(20**10000)
5788
>>> sys.getsizeof(20**1000000)
576284
>>> sys.getsizeof(20**100000000) # Il calcolo impiega molto tempo....
57625732
```

Questa caratteristica non è scontata, visto che non tutti i linguaggi di programmazione gestiscono gli interi come fa Python. Ad esempio, con i tipi nativi di alcuni importanti linguaggi di più basso livello, come C, C++ o Java, non è possibile rappresentare tutti gli interi, ma solamente quelli più grandi di un dato valore minimo e più piccoli di un dato valore massimo.

Chi si disinteressa di queste problematiche rischia di commettere dei grossi errori, anche effettuando una banale somma tra due interi. Consideriamo, ad esempio, il seguente codice C++:

```
$ cat foo.cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    long long int a = 9223372036854775807;
    long long int b = -(a + 1);
    cout << showpos;
    cout << left << setw(5) << "a" << setw(5) << "->" << a << endl;
    cout << left << setw(5) << "a+1" << setw(5) << "->" << a + 1 << endl << endl;
    cout << left << setw(5) << "b" << setw(5) << "->" << b << endl;
    cout << left << setw(5) << "b-1" << setw(5) << "->" << b - 1 << endl;
}
```

Questo assegna il valore `9223372036854775807` alla variabile `a`, poi assegna il valore -

(a+1) alla variabile b, poi stampa a, a+1, b e b-1. Ecco cosa accade se lo compiliamo e poi lo eseguiamo:

```
$ g++ -o foo foo.cpp # Compiliamo
```

```
$ ./foo # Eseguiamo
```

```
a -> +9223372036854775807
```

```
a+1 -> -9223372036854775808
```

```
b -> -9223372036854775808
```

```
b-1 -> +9223372036854775807
```

Come possiamo vedere, $9223372036854775807 + 1$ dà un risultato negativo, mentre $-9223372036854775808 - 1$ dà un risultato positivo. Questo errore è dovuto al fatto che alla variabile a è stato assegnato il valore massimo che può avere un intero con segno, codificato con 64 bit, per cui, sommando un altro intero a questo valore massimo, otteniamo un valore troppo grande per essere rappresentato all'interno dello spazio disponibile per la sua memorizzazione. In questo caso si dice che si ha un *overflow*.

Anche se può sembrare incredibile, tantissimi programmatori commettono errori di questo tipo (talvolta senza neppure accorgersene), dovuti principalmente al fatto che non hanno sufficienti conoscenze del sistema hardware/software sul quale lavorano.

Torniamo a noi, ovvero a Python. Abbiamo detto che il tipo `int` consente di rappresentare interi di qualunque valore, per cui le somme e le sottrazioni che in C++ hanno generato un overflow con Python vengono eseguite in modo corretto:

```
>>> a = 9223372036854775807
```

```
>>> a + 1
```

```
9223372036854775808
```

```
>>> b = -(a + 1)
```

```
>>> b - 1
```

```
-9223372036854775809
```

A questo punto probabilmente ci stiamo chiedendo perché mai dovremmo preoccuparci dell'overflow, se questo problema perseguita solamente gli sfortunati che programmano con altri linguaggi. Come possiamo vedere, non è esattamente così:

```
>>> import numpy # Libreria di terze parti utilizzata per il calcolo scientifico
```

```
>>> a, b = numpy.array([9223372036854775807, 1])
```

```
>>> a
```

```
9223372036854775807
```

```
>>> b
1
>>> a + b
-9223372036854775808
```

Infatti potrebbe accadere che una libreria di terze parti non utilizzi il tipo `int` di Python per operare con gli interi. Nel caso precedente, ad esempio, abbiamo utilizzato la famosa libreria NumPy e abbiamo creato un array di interi. NumPy, però, ha convertito implicitamente gli elementi della lista in oggetti di tipo `numpy.int64`:

```
>>> type(a)
<class 'numpy.int64'>
```

Il tipo `numpy.int64` codifica gli interi con 64 bit ed è questo il motivo per cui questi interi non possono assumere un valore più grande di 9223372036854775807 e più piccolo di -9223372036854775808. Infatti, se n è il numero di bit che possiamo utilizzare per codificare un intero con segno, allora il massimo valore rappresentabile è dato da $2^{n-1} - 1$, mentre il minimo è dato da -2^{n-1} .

NOTA

Vediamo come vengono codificati gli interi con segno avendo a disposizione un numero limitato di bit. Se, ad esempio, utilizziamo $n = 3$ bit per rappresentare gli interi, allora possiamo avere solamente le seguenti codifiche: 000, 001, 010, 011, 100, 101, 110, 111. Il primo bit viene utilizzato per indicare il segno del numero: 0 indica che il numero ha segno positivo, 1 che ha segno negativo. In questo modo 000 rappresenta il numero decimale 0, 001 il numero 1, 010 il numero 2 e 011 il numero 3, in accordo con la formula generale secondo la quale il massimo valore rappresentabile è dato da $2^{n-1} - 1$. I codici che iniziano con 1 rappresentano numeri negativi. La più semplice codifica dei numeri negativi consiste nell'assegnare al codice 100 il valore decimale 0, al codice 101 il valore -1, al codice 110 il valore -2 e al codice 011 il valore -3. Questo tipo di codifica è detto *in segno e modulo* e viene utilizzato di rado. Si preferisce utilizzare una codifica detta *in complemento a due*, nella quale, quando il numero è negativo (primo bit uguale a 1), si invertono tutti i bit (gli 1 diventano 0 e viceversa), si somma poi 1 e si calcola il valore del numero su tutti gli n bit (primo compreso). Questo valore, visto che il numero è negativo, viene cambiato di segno. Ad esempio, il codice 100 corrisponde al numero decimale -4. Infatti invertito diventa 011, sommato a 1 diventa

nuovamente 100, per cui il suo valore decimale è 4 e, in definitiva, cambiato di segno diventa -4. Il codice 101 invertito diventa 010, poi sommato a 1 diventa 011, a cui corrisponde il numero 3, ovvero -3. Il codice 110 diventa 001, sommato a 1 diventa 010, a cui corrisponde il numero 2, ovvero -2. Infine il codice 111 diventa 000, sommato a 1 diventa 001, a cui corrisponde il numero 1, quindi -1. Rispetto alla codifica in segno e modulo vi è una unica rappresentazione dello zero:

000 → 0 001 → 1 010 → 2 011 → 3 100 → -4 101 → -3 110 → -2 111 → -1

Il minimo valore rappresentabile è dato, quindi, da -2^{n-1} . Questo spiega anche perché nei nostri esempi l'overflow di una unità ha generato in un caso l'opposto positivo e nell'altro quello negativo. Sommando, infatti, una unità al massimo positivo, otteniamo l'estremo negativo: $011 + 001 = 100$, ovvero -4. Viceversa, sottraendo una unità all'estremo negativo otteniamo l'estremo positivo: $100 - 001 = 100 + 111 = 011$, ovvero 3. Quindi, riassumendo, quando si va in overflow positivo di m , anziché ottenere $(\max + m)$ si ottiene $(\min - m + 1)$, mentre quando si va in overflow negativo di m , anziché ottenere $(\min - m)$ si ottiene $(\max + m - 1)$.

Morale della favola, se lavoriamo con Python utilizzando librerie di terze parti, allora dobbiamo conoscere il tipo con il quale la libreria rappresenta gli interi. Se questo non è `int` e gli interi sono codificati con un numero finito n di bit, allora dobbiamo prestare attenzione, perché l'insieme dei numeri rappresentabili è limitato superiormente da $2^{n-1}-1$ e inferiormente da -2^{n-1} , e quindi, superando questi limiti, si va in overflow.

NOTA

Il modulo `ctypes` della libreria standard ci consente di utilizzare dei tipi di dato compatibili con i corrispondenti tipi nativi del C:

```
>>> import ctypes
>>> a = ctypes.c_int(-22)
>>> b = ctypes.c_ulonglong(10000)
>>> c = a.value * b.value
>>> type(c)
<class 'int'>
>>> c
-220000
```

Numeri floating point

Come ormai sappiamo, i letterali *floating point* si distinguono dagli interi per la presenza del punto decimale. Vi sono anche altre rappresentazioni, nelle quali il numero è indicato in notazione esponenziale, con l'esponente seguito dai caratteri `e` o `E`:

```
>>> 400., 400.0, 4e2, 4E2, 4e-2 # I letterali 4e2 e 4E2 corrispondono a 4 * 10**2
(400.0, 400.0, 400.0, 400.0, 0.04)
```

NOTA

Visto che una cifra seguita da un punto è utilizzata per rappresentare il letterale di un numero floating-point, non possiamo accedere a un attributo di un numero intero tramite il suo letterale in modo classico:

```
>>> 7.bit_length()
File "<stdin>", line 1
7.bit_length()
^
SyntaxError: invalid syntax
```

In questo caso siamo costretti a racchiudere il letterale intero tra le parentesi tonde:

```
>>> (7).bit_length()
3
```

Gli operatori di divisione si comportano in modo analogo a quanto visto per i numeri interi. Ricordiamo che la *divisione floor* restituisce un `float` quando almeno uno dei suoi operandi è di tipo `float`:

```
>>> -11.0 / 2 # Divisione vera, restituisce un `float`
-5.5
>>> -11.0 // 2 # Arrotonda all'intero inferiore e restituisce un float
-6.0
>>> 10.0 // 2 # Restituisce sempre un `float` quando uno dei due operandi è un `float`
5.0
```

Conversioni

Iniziamo questa sezione con una precisazione: l'operazione di conversione di un floating-point all'intero inferiore è indicata con il termine *floor*; la conversione di un floating point a un intero mediante perdita delle sue cifre decimali è chiamata *troncamento*; la conversione di un floating point all'intero più vicino è detta *arrotondamento*.

Conversioni esplicite

La classe built-in `float` converte un intero o una stringa in `float`:

```
>>> float(33), float("33"), float("33.0"), float('0.33e2')
(33.0, 33.0, 33.0, 33.0)
```

La classe built-in `int` consente di passare da floating point a intero effettuando un troncamento:

```
>>> int(-33.6), int(33.6) # Troncamento
(-33, 33)
```

La funzione built-in `round()` arrotonda un `float` all'intero più vicino:

```
>>> round(11.60), round(-11.30) # Arrotondamento all'intero più vicino
(12, -11)
>>> int(11.60), int(-11.30) # Troncamento, ovvero perdita della parte decimale
(11, -11)
```

La funzione built-in `round()` accetta un secondo argomento che indica la precisione:

```
>>> round(-33.446, 2), round(33.446, 2)
(-33.45, 33.45)
```

La conversione di un `float` all'intero inferiore può essere fatta con la funzione `math.floor()`.

Ecco quindi il quadro completo:

```
>>> import math
>>> math.floor(-11.6), math.floor(11.6) # Conversione all'intero inferiore
(-12, 11)
>>> int(-11.6), int(11.6) # Troncamento
(-11, 11)
>>> round(-11.6), round(11.6) # Arrotondamento
(-12, 12)
```

Conversioni implicite

Nelle espressioni in cui, oltre all'operando `floating point`, vi è un operando intero, quest'ultimo viene convertito in `float` e il risultato è a sua volta un `float`:

```
>>> 44 * 2.0, 44 / 2.0, 44 + 2.0, 44 - 2.0 # Conversione implicita al tipo più specializzato
(88.0, 22.0, 46.0, 42.0)
```

Aritmetica dei numeri *floating-point*

Un programmatore alle prime armi potrebbe pensare che il tipo `float` ci consenta di lavorare con i numeri reali. Questa è solo un'illusione, perché un calcolatore è in grado di rappresentare solamente un insieme finito di numeri, detti *numeri macchina*. Spesso non ci accorgiamo di questo limite architetturale, perché in alcuni casi i calcoli producono lo stesso risultato che si ottiene operando con i numeri reali:

```
>>> 2.0 ** 4
16.0
>>> 3.0 / 2
1.5
```

In altri casi, però, ci può essere una differenza:

```
>>> 0.1 * 3
0.30000000000000004
>>> 0.3
0.3
```

Per quanto “piccola” possa essere, questa differenza può portarci a commettere grossi errori logici:

```
>>> 0.1 * 3 == 0.3
False
>>> 10.1 + 10.2 == 20.3
False
```

Talvolta i risultati potrebbero essere anche molto distanti dalle attese:

```
>>> a = 2.0e200 # 2.0 * 10**200
>>> b = 2.0e900 # 2.0 * 10**900
>>> b - b == 0
False
>>> a - a == 0
True
>>> a * b == b
True
```



```
>>> a * b - b == a * b - b
False
```

Per poter capire quali sono le problematiche alle quali si va incontro eseguendo delle operazioni con i `float`, partiamo da lontano, ovvero dal concetto di floating-point e dalla sua implementazione secondo lo standard IEEE 754. Nelle prossime due sezioni (intitolate *La notazione floating-point* e *IEEE Standard for Binary Floating-Point Arithmetic*) vedremo, quindi, della teoria, ma non spaventiamoci, perché ritorneremo subito alla pratica, a partire dalla sezione *Valori massimo e minimo rappresentabili*.

Visto che in tutti i linguaggi di programmazione più diffusi il tipo `float` è implementato in accordo con questo standard, i concetti che tratteremo sono di validità generale.

La sezione è molto specifica, per cui, se in questo momento tutti questi dettagli non vi interessano, potremo riprendere questi argomenti in un secondo momento. Teniamo comunque presente che, facendo un piccolo sforzo per studiare le prossime pagine di teoria, potremo evitare che nei nostri programmi (scritti in Python o in qualunque altro linguaggio) vi siano grossi errori logici.

La notazione floating-point

Nelle sezioni precedenti abbiamo visto come con Python sia possibile rappresentare qualsiasi numero intero, mentre questo non è possibile con i tipi nativi di altri importanti linguaggi di programmazione. Abbiamo anche detto che la maggiore flessibilità di Python è dovuta al fatto che usa un numero variabile di byte per rappresentare gli interi.

I `float`, invece, sono codificati con un numero fisso di byte, infatti si basano sui `double` (*doppia precisione*) del linguaggio C, i quali sono implementati in accordo con lo standard IEEE 754 che, come vedremo tra breve, definisce delle convenzioni per rappresentare con una sequenza *finita* di byte la notazione floating-point dei numeri. Come possiamo verificare, infatti, la dimensione in byte dei `float` è costante, ovvero non aumenta all'aumentare del valore assoluto del numero:

```
>>> sys.getsizeof(2.0)
24
>>> sys.getsizeof(2.0 ** 10)
24
>>> sys.getsizeof(2.0 ** 100)
24
>>> sys.getsizeof(2.0 ** 1000)
```

Ma andiamo per gradi e vediamo in cosa consiste la notazione floating-point. Un numero reale r può essere scritto nella forma $r = mb^q$, dove $m \in \mathbb{R}$ è detta mantissa, b è detta base del sistema di numerazione e $q \in \mathbb{Z}$ è detto esponente. Ad esempio, secondo questa notazione il numero $r = 344.75$ può essere scritto in base 10 come $r = 3.4475 \cdot 10^2$. In questo caso $m = 3.4475$, $b = 10$, $q = 2$.

Questa notazione è detta *floating-point* (*virgola mobile*) e non è unica, ovvero consente di rappresentare un numero con diversi valori di m , b e q . Infatti possiamo scrivere $r = 0.34475 \cdot 10^3$ o anche $r = 34.475 \cdot 10^1$.

Passiamo adesso a qualche esempio di rappresentazione floating-point in base 2, che più ci interessa, visto che è quella utilizzata dai calcolatori. Vediamo prima come convertire un numero con parte frazionaria da rappresentazione decimale a binaria. Consideriamo, ad esempio, $r = 100.11101$. La prima cifra alla sinistra del punto ha indice 0, la seconda ha indice 1, la terza ha indice 2, mentre la prima cifra alla destra del punto ha indice -1, la seconda -2 e così via. La conversione da rappresentazione binaria a decimale si ottiene utilizzando gli indici come esponenti della base:

$$100.11101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} = 4.90625$$

La [Figura 2.1](#) schematizza quanto appena detto.

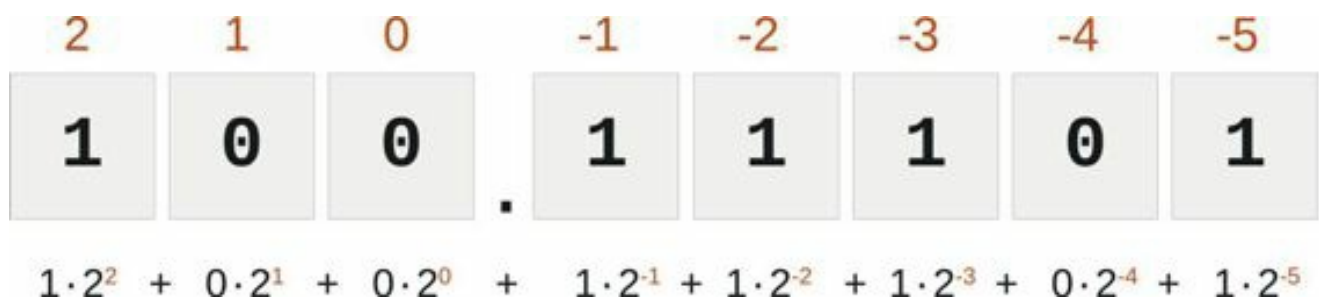


Figura 2.1 - Conversione del numero $r = 100.11101$ dalla rappresentazione binaria a quella decimale.

Se utilizziamo la notazione floating-point in base 2, allora possiamo scrivere $r = 10.011101 \cdot 2^1$, con $m = 10.011101$ e $q = 1$, o anche $r = 1.0011101 \cdot 2^2$, con $m = 1.0011101$ e $q = 2$.

La rappresentazione di r si dice *normalizzata* quando la mantissa, prima del punto decimale, ha una sola cifra e questa è diversa da zero. Questa cifra è compresa tra 1 e 9 se la base è 10 ed è sempre 1 se la base è 2. In questo caso le

cifre di m che seguono il punto sono dette *parte frazionaria* o *cifre significative* e vengono indicate con f . Ad esempio, in base 10 la rappresentazione normalizzata di $r = 77.15$ è $r = 7.715 \cdot 10^1$, con $m = 7.715$, $f = 0.715$ e $q = 1$, mentre quella di $r = 0.000431$ è $r = 4.31000 \cdot 10^{-4}$. In base 2 la rappresentazione normalizzata di $r = 100.11101$ è $r = 1.0011101 \cdot 2^2$, con $f = 0.0011101$ e $q = 2$.

Fissata la base b , ogni numero reale r in rappresentazione floating-point normalizzata è univocamente determinato dalla coppia (q, m) . Nella notazione floating-point in base 2, inoltre, visto che la prima cifra di m è sempre pari a 1, per rappresentare r in modo univocamente determinato è sufficiente conoscere la coppia (q, f) e il segno s (+ o -) del numero (visto che questo non è compreso in f): $r \Leftrightarrow (s, q, f)$.

IEEE Standard for Binary Floating-Point Arithmetic

Lo standard IEEE per il calcolo in virgola mobile con base 2, indicato con IEEE 754, è stato istituito nel 1985 dallo IEEE (Institute of Electrical and Electronics Engineers). L'ultima versione, pubblicata nel 2008, estende lo standard rendendolo indipendente dalla base di numerazione.

Questo standard stabilisce come rappresentare, con un numero finito di byte, sia i numeri floating-point normalizzati sia alcuni casi speciali. In altre parole, stabilisce sia il numero di bit da utilizzare per rappresentare la mantissa, l'esponente e il segno, sia il significato delle codifiche.

NOTA

Per stabilire il numero di bit da assegnare alla parte frazionaria e all'esponente si è dovuto trovare un compromesso tra range di valori ammissibili e accuratezza. Infatti, maggiore è il numero di bit che si assegnano alla parte frazionaria e maggiore è l'accuratezza, mentre maggiore è il numero di bit che si assegnano all'esponente, più grande è il massimo valore rappresentabile.

Ad esempio, nel caso di Python il tipo `float` è mappato sui `double` del linguaggio C, per cui ogni numero floating-point viene codificato con 8 byte. In questo caso lo standard prescrive di utilizzare un bit per rappresentare il segno (0 significa segno positivo, 1 significa segno negativo), 11 bit per l'esponente e 52 bit per la parte frazionaria, come mostrato in [Figura 2.2](#).



Figura 2.2 - Significato dei bit di un floating-point in doppia precisione secondo lo standard IEEE 754.

Per indicare il segno dell'esponente non è stato assegnato alcun bit perché a q viene sottratto un *bias*:

$$r = (-1)^s(1 + f)b_{q-bias} = (-1)^s(1 + f)b^e \quad (2.1)$$

Il *bias* viene scelto in modo tale che $q - bias$ possa assumere valori sia negativi sia positivi e nel caso della doppia precisione è pari a 1023. Poiché q è codificato con 11 bit, il suo valore va da 0 (undici bit a zero) a 2047 (undici bit a uno). Come vedremo tra breve, lo standard prevede dei casi speciali per questi due valori, per cui in realtà i valori di q che possono essere utilizzati per rappresentare i floating-point normalizzati vanno da 1 a 2046. Questo significa che nella formula 2.1 l'esponente che si ricava sulla base del valore di q può assumere solo i valori interi che vanno da quello minimo:

$$e_{min} = q_{min} - bias = 1 - 1023 = -1022$$

fino al massimo:

$$e_{max} = q_{max} - bias = 2046 - 1023 = 1023$$

Passiamo alla pratica e vediamo finalmente cosa comporta tutto ciò.

Valori massimo e minimo rappresentabili

Sulla base di quanto appreso nella sezione precedente, siamo in grado di fare alcuni conti. Vediamo, innanzitutto, qual è il massimo valore rappresentabile da un float. A tale scopo consideriamo la Figura 2.3. Questa mostra la codifica del più grande numero floating-point che possiamo ottenere. Infatti q assume il suo valore massimo (2046) e anche la parte frazionaria (52 bit a 1).



Figura 2.3 - Rappresentazione binaria in doppia precisione del massimo numero floating-point rappresentabile.

Il valore della parte frazionaria è dato da:

```
>>> f = sum(2**(-i) for i in range(1, 53)) # 1/2 + 1/4 + 1/8 + 1/16 + ... + 1/(2 ** 52)
```

mentre il valore dell'esponente risulta $e = q - bias = 1023$. Quindi, sulla base della formula 2.1, il massimo numero floating-point rappresentabile risulta:

```
>>> (1 + f) * 2**1023
1.7976931348623157e+308
```

L'attributo `sys.float_info` ci dà conferma della correttezza dei nostri conti:

```
>>> sys.float_info.max
1.7976931348623157e+308
```

La [Figura 2.4](#) mostra la codifica del più piccolo floating-point in valore assoluto. Infatti q assume il suo valore minimo (1) e anche la parte frazionaria (52 bit a 0).



Figura 2.4 - Rappresentazione binaria in doppia precisione del massimo numero floating-point rappresentabile.

Quindi $e = q - bias = -1022$, $f = 0$ e $m = 1$, per cui, in base alla formula 2.1, il più piccolo numero floating-point normalizzato in valore assoluto è dato da:

```
>>> 1 * 2**(-1022)
2.2250738585072014e-308
```

Infatti:

```
>>> sys.float_info.min
```

Lo standard IEEE 754-2008 consente di rappresentare dei numeri più piccoli, in valore assoluto, del minimo valore normalizzato rappresentabile. A questi numeri, caratterizzati dall'avere $q = 0$ e $f \neq 0$, non viene aggiunto implicitamente il valore 1 alla parte frazionaria e per questo motivo sono detti *denormalizzati* (*subnormal*): $r = (-1)^s f \cdot 2^{-1022} = (-1)^s f \cdot \text{sys.float_info.min}$.

In [Figura 2.5](#) è illustrato qualche esempio di codifica di numeri denormalizzati.



Figura 2.5 - Esempio di codifica di alcuni numeri denormalizzati.

Il più piccolo numero rappresentabile, in valore assoluto, è quindi quello denormalizzato avente il solo ultimo bit a 1:

```
>>> min_den = 1/2**52 * sys.float_info.min
>>> min_den
5e-324
```

Riassumendo, in questa sezione abbiamo visto sia che l'insieme dei numeri macchina è limitato (superiormente da `sys.float_info.max` e inferiormente da `-sys.float_info.max`), sia che il più piccolo numero macchina (in valore assoluto e in doppia precisione) è `5e-324`.

Underflow e overflow

Abbiamo appena detto che il più piccolo numero rappresentabile, in valore assoluto, risulta:

```
>>> min_den = 1/2**52 * sys.float_info.min
>>> min_den
5e-324
```

Quindi non possiamo ottenere alcun numero positivo inferiore a questo:

```
>>> min_den / 2
```

Ovvero, non esiste alcun numero macchina m tale che $0 < m < 1/2^{52} * \text{sys.float_info.min}$. In altre parole, quando il risultato di un'operazione dovrebbe essere compreso tra 0 e $\text{min_den}/2$, otteniamo un errore di *underflow*, perché il risultato viene approssimato a zero.

In generale, dati due numeri m_1 e m_2 appartenenti a M , non è detto che esista un terzo numero $m \in M$ tale che $m_1 < m < m_2$. In termini tecnici si dice che l'insieme dei numeri macchina non è denso.

Se M è limitato superiormente e inferiormente, allora dobbiamo aspettarci delle situazioni di overflow. Vediamo cosa accade se proviamo a ottenere un floating-point di valore superiore a quello massimo:

```
>>> m = sys.float_info.max
>>> a = 1.0 * 2 ** 100
>>> m
1.7976931348623157e+308
>>> a
1.2676506002282294e+30
>>> m + a
1.7976931348623157e+308
```

Come possiamo vedere, il risultato è ancora il massimo:

```
>>> m + a == m
True
```

Questo risultato è dovuto alla cosiddetta *perdita di cifre significative*, causata dall'algoritmo utilizzato per effettuare la somma di due floating-point. Vediamo subito un esempio pratico che mostra come viene effettuata la somma. Supponiamo di voler sommare i numeri $r_1 = 5$ e $r_2 = 6.25$ che in base 2 normalizzata risultano essere, rispettivamente, $r_1 = 1.01 \cdot 2^2$ e $r_2 = 1.1101 \cdot 2^1$. Supponiamo, inoltre, per semplicità, di avere solamente 4 bit per rappresentare la parte frazionaria. Per poter sommare i due numeri dobbiamo, innanzitutto, eguagliare l'esponente più piccolo a quello più grande. Per fare ciò la mantissa del numero con esponente più piccolo viene traslata a destra di un numero di posizioni pari alla differenza degli esponenti, come mostrato in [Figura 2.6](#).



Figura 2.6 - L'incremento dell'esponente in un floating-point è ottenuto mediante traslazione a destra della mantissa.

Come possiamo osservare, il fatto di avere solamente 4 bit a disposizione ha causato la perdita di una cifra significativa, per cui abbiamo introdotto un errore ancor prima di aver effettuato la somma. Infatti sommeremo al numero 5 non più 11.1010 (3.625 in base 10), ma 11.10 (3.5 in base 10), come mostrato in [Figura 2.7](#).

Il risultato della somma è $10.0010 \cdot 2^2$, per cui per normalizzarlo dobbiamo traslare la mantissa a destra di una cifra, e incrementare di una unità l'esponente. In questo modo perdiamo, ancora una volta, una cifra significativa, ma non introduciamo alcun errore, visto che questa volta la cifra è uno zero in posizione finale. Il risultato della somma è, quindi, 8.5 anziché 8.625.

$$\begin{array}{r}
 \boxed{1} \boxed{.} \boxed{0} \boxed{1} \boxed{0} \boxed{0} \cdot 2^2 \quad + \\
 \boxed{0} \boxed{.} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \cdot 2^2 \quad = \\
 \hline
 \boxed{1} \boxed{0} \boxed{.} \boxed{0} \boxed{0} \boxed{1} \boxed{0} \cdot 2^2 \quad = \\
 \boxed{1} \boxed{.} \boxed{0} \boxed{0} \boxed{0} \boxed{1} \boxed{0} \cdot 2^3
 \end{array}$$

Figura 2.7 - Secondo step nella somma tra due numeri floating-point in rappresentazione binaria.

NOTA

In realtà lo standard IEEE 754 prevede l'utilizzo di alcuni bit aggiuntivi, detti *extra bits*, che durante i calcoli intermedi consentono di ridurre gli errori dovuti alla perdita di cifre significative.

Il fatto che $m + a$, con $m = \text{sys.float_info.max}$ e $a = 1.0 \cdot 2^{100}$ dia come risultato m è legato

proprio a questo. Infatti, quando la mantissa di a viene traslata al fine di eguagliare il suo esponente a quello di m , a ogni step il bit di valore 1 si sposta a destra di una posizione, finché non viene perso al 53° spostamento, come mostrato in [Figura 2.8](#).



Figura 2.8 - Perdita delle cifre significative di a nella somma $m+a$, con $a=1.0 \cdot 2^{100}$ e $m=\text{sys.float_info.max}$.

Il risultato di tutto ciò è che, alla fine, come mostrato in [Figura 2.9](#), viene sommato 0 a m , e questo è il motivo per cui il risultato di $m+a$ è m .



Figura 2.9 - Risultato della somma binaria di $m+a$, dopo aver eguagliato l'esponente di a a quello di m .

Cosa accade se la differenza tra l'esponente di m e quello di a non è tanto grande da causare la perdita dei bit di valore 1 di a ? Accade che nella somma di [Figura 2.9](#) vi è un bit con valore 1 nella parte frazionaria di a , per cui il risultato della somma è un numero superiore al valore massimo. Questa è quindi una condizione di *overflow*, e il risultato dell'operazione viene indicato con inf , che significa *infinito*:

```
>>> m + 1.0 * 2 ** (1023-53)
inf
```

Se proviamo a effettuare un elevamento a potenza utilizzando un valore dell'esponente superiore a quello massimo, non otteniamo `inf` ma un errore:

```
>>> 2.0 ** 1024
Traceback (most recent call last):
...
OverflowError: (34, 'Numerical result out of range')
```

Per ottenere `inf` in modo esplicito al di fuori di una operazione, possiamo passare alla classe `float` le stringhe `'inf'` o `'infinity'`, indifferentemente con i caratteri maiuscoli o minuscoli:

```
>>> float('inf')
inf
>>> float('infinity')
inf
>>> float('InFiNiTy')
inf
```

Lo stesso risultato si ottiene in modo meno esplicito, utilizzando un letterale che rappresenta un `float` di valore superiore a quello massimo:

```
>>> 2.0e900 # 2.0 * 10**900
inf
```

Visto che *esplicito è meglio che implicito*, la prima soluzione è senza dubbio la migliore. Per lo stesso motivo, la seguente divisione non dà `inf` come risultato, ma un errore di overflow:

```
>>> 2**1027 / 2**3
Traceback (most recent call last):
...
OverflowError: integer division result too large for a float
```

Abbiamo detto, infatti, che in Python 3 la divisione restituisce sempre un `float`, anche quando i due operandi sono interi e il risultato potrebbe essere un `int`. Questa conversione è implicita e questo è il motivo per cui viene generato un errore di overflow.

Si osservi che `inf` non è una stringa ma un `float`:

```
>>> result = 2 * 2.0**1023
>>> result
```

```
inf
>>> type(result)
<class 'float'>
```

Possiamo quindi utilizzarlo come operando in una operazione tra `float`:

```
>>> i = float('inf')
>>> i
inf
>>> -i
-inf
>>> 2.0 * i
inf
>>> 2 ** -i
0.0
>>> i / i
Nan
>>> i - i
nan
```

Ecco un'altra sorpresa: `nan` significa *not a number*, ed è anch'esso un `float`:

```
>>> type(i/i)
<class 'float'>
```

Può essere ottenuto anche passando la stringa `'nan'` alla classe `float`:

```
>>> float('nan')
nan
>>> float('NaN')
nan
```

In qualsiasi operazione in cui compare `nan` come operando, il risultato è sempre `nan`:

```
>>> n = i / i
>>> n * 2
nan
>>> n + 1
nan
>>> n * i
nan
```

Inoltre, visto che `nan` non è un numero, non può essere uguale a nessun altro numero. Potrebbe sorprendere che non sia uguale neppure a se stesso:

```
>>> nn = float('nan')
>>> nn
nan
```

```
>>> nn is nn
True
>>> nn == nn
False
```

NOTA

Il massimo valore che può assumere q , 2047, è utilizzato per rappresentare `inf` e `nan`, e questo è il motivo per cui l'esponente $q - bias$ nei floating-point normalizzati ha come valore massimo 1023 piuttosto che 1024. È la parte frazionaria che differenzia `inf` da `nan`: nel caso di `inf` si ha $f = 0$, mentre nel caso di `nan` si ha $f \neq 0$.

Utilizzando l'operatore di confronto non è possibile, quindi, sapere se un `float` è `nan`. Per poterlo fare dobbiamo utilizzare la funzione `math.isnan()`:

```
>>> import math
>>> math.isnan(nn)
True
```

Esiste anche la funzione `math.isinf()`, la quale ci dice se un `float` è `inf`:

```
>>> math.isinf(float('inf'))
True
```

A questo punto il risultato di parte delle istruzioni viste all'inizio della sezione dovrebbe essere chiaro. Consideriamo, ad esempio, le seguenti:

```
>>> a = 2.0e200 # 2.0 * 10**200
>>> b = 2.0e900 # 2.0 * 10**900
>>> b - b == 0
False
```

L'etichetta `b` fa riferimento a `inf`:

```
>>> b
inf
```

per cui `b - b` produce come risultato `nan`, che giustamente è diverso da 0:

```
>>> result = b - b
>>> result
nan
>>> result == 0
```

False

L'etichetta `a` fa invece riferimento a un `float` di valore inferiore a quello massimo:

```
>>> a
2e+200
```

Per cui, giustamente, `a - a` è uguale a 0:

```
>>> a - a == 0
True
```

Il risultato di `a * b` è `inf`, per cui l'espressione `a * b == inf` equivale a `inf == inf`, e questa dà come risultato `True`:

```
>>> a * b == b
True
```

Infine abbiamo:

```
>>> a * b - b == a * b - b
False
```

Questa equivale a `nan == nan`, che produce `False`.

Precisione macchina

Abbiamo visto che l'insieme dei numeri macchina è limitato superiormente e inferiormente, e questo può portare a problemi di overflow. Altri errori derivano dal fatto che M non è denso, per cui non possiamo rappresentare in modo esatto tutti i numeri reali. Ad esempio, non possiamo rappresentare i numeri irrazionali (π , $\sqrt{2}$ ecc.) poiché la loro parte frazionaria ha un numero infinito di cifre. Per lo stesso motivo, non possiamo rappresentare i numeri periodici. In questo ultimo caso dobbiamo prestare particolare attenzione, perché un numero che non è periodico in base 10 può esserlo in base 2. L'esempio classico è il numero decimale 0.1, che in base 2 è periodico (0.000110011...). Questo è il motivo per cui otteniamo:

```
>>> 0.1 * 3 == 0.3
False
>>> 0.1 + 0.2 == 0.3
False
```

NOTA

Se vogliamo lavorare in rappresentazione decimale, possiamo utilizzare il modulo `decimal` della libreria standard:

```
>>> from decimal import Decimal
>>> Decimal('0.1') * Decimal('3') == Decimal('0.3')
True
>>> Decimal('0.1') + Decimal('0.2') == Decimal('0.3')
True
>>> import numbers
>>> isinstance(Decimal('33.3'), numbers.Number)
True
```

Il modulo `decimal` ha anche un range di valori più ampio di quello dei `float`:

```
>>> import decimal
>>> decimal.MAX_EMAX # Massimo esponente
999999999999999999
>>> decimal.Decimal('10e9999999999999999')
Decimal('1.0E+999999999999999999')
```

La rappresentazione decimale dei numeri ha anch'essa i suoi limiti:

```
>>> Decimal(str(1/3)) * Decimal('1.5')
Decimal('0.49999999999999995')
```

La libreria standard ci consente di lavorare anche con l'aritmetica dei *numeri razionali*:

```
>>> from fractions import Fraction
>>> Fraction(1, 3) * Fraction(3, 2)
Fraction(1, 2)
>>> import numbers
>>> isinstance(Fraction(1, 3), numbers.Number)
True
```

In base a tutto ciò, non dovremmo quindi stupirci del fatto che, in generale, le proprietà associativa e distributiva non sono rispettate:

```
>>> (0.1 + 0.2) + 0.3 == 0.1 + (0.2 + 0.3)
False
>>> >>> 0.3*(0.1 + 0.2) == 0.3*0.1 + 0.3*0.2
False
```

A questo punto è lecito chiedersi come poter comparare i floating-point. Se

conosciamo a priori l'ordine di grandezza dei numeri, allora possiamo fare il confronto utilizzando come metro di giudizio l'*errore assoluto*. In altre parole, possiamo decidere che per i nostri scopi i numeri sono uguali quando lo sono le loro prime n cifre significative:

```
>>> def isAlmostEqual(a, b, delta=1e-5):
...     return abs(a - b) < delta
...
>>> isAlmostEqual(0.000123, 0.000122)
True
```

Se, invece, non conosciamo a priori l'ordine di grandezza dei numeri che vogliamo confrontare, allora dobbiamo fare la comparazione valutando l'*errore relativo*:

```
>>> def isAlmostEqual(a, b, delta=1e-5):
...     return abs((a-b)/b) < delta # Con b diverso da 0...
...
>>> a = 0.1 + 0.2
>>> b = 0.3
>>> isAlmostEqual(a, b)
True
```

Dato un numero reale α , il numero macchina che lo rappresenta è, in generale, un'approssimazione, che indicheremo con $\bar{\alpha}$. L'errore relativo che si commette nell'effettuare questa approssimazione è sempre inferiore o uguale a un numero macchina ϵ :

$$\left| \frac{\bar{\alpha} - \alpha}{\alpha} \right| \leq \epsilon$$

Questo numero è detto *precisione macchina* e nel nostro caso (doppia precisione) è il seguente:

```
>>> sys.float_info.epsilon
2.220446049250313e-16
```

L'errore relativo tra due numeri non può mai essere inferiore a $\epsilon/2$, per cui non ha senso utilizzare dei valori di δ inferiori a questo limite:

```
>>> isAlmostEqual(a, b, sys.float_info.epsilon)
True
>>> isAlmostEqual(a, b, sys.float_info.epsilon/2)
False
```

Metodi dei numeri floating point

Iniziamo con il mostrare il metodo `float.as_integer_ratio()`, già incontrato nel [Capitolo 1](#):

```
>>> num = 3.5
>>> num.as_integer_ratio() # Restituisce una coppia di interi il cui rapporto è `num`
(7, 2)
```

Abbiamo visto quali sono i problemi dell'aritmetica finita, per cui non stupiamoci se alcuni risultati non sono quelli che ci aspettiamo:

```
>>> 0.1.as_integer_ratio() # In base a quanto visto prima, il risultato non sarà (1, 10)
(3602879701896397, 36028797018963968)
```

Il metodo `float.is_integer()` restituisce `True` se il `float` ha una parte decimale nulla:

```
>>> 10.5.is_integer()
False
>>> 10.0.is_integer()
True
```

Dato un `float` che indichiamo con `num`, il metodo `num.hex()` restituisce una stringa contenente una rappresentazione esadecimale di `num`:

```
>>> num = 7.5
>>> num.hex()
'0x1.e000000000000p+2'
```

Il metodo `float.fromhex()` prende una stringa contenente un floating-point in rappresentazione esadecimale e restituisce il corrispondente `float`:

```
>>> float.fromhex('0x1.e000000000000p+2')
7.5
```

Numeri complessi

I *letterali complessi* si distinguono da quelli interi per la presenza del carattere `j` (maiuscolo o minuscolo) che caratterizza la parte immaginaria:

```
>>> num = 44 + 1j # Creiamo un numero complesso con parte reale `44` e parte immaginaria `1`
```

La parte reale e quella immaginaria di un numero complesso sono oggetti di tipo `float`:


```
>>> num.real, type(num.real), num.imag, type(num.imag)
(44.0, <class 'float'>, 1.0, <class 'float'>)
```

Il metodo `complex.conjugate()` restituisce il coniugato di un numero:

```
>>> num.conjugate() # Restituisce il coniugato di `num`
(44-1j)
```

La funzione built-in `abs()` restituisce il modulo del numero complesso passato come argomento:

```
>>> abs(num)
44.01136216933077
```

Nelle operazioni dove compaiono i numeri complessi il risultato è sempre un numero complesso, anche quando la parte immaginaria è nulla:

```
>>> num * num.conjugate() # num.imag**2 + num.real**2
(1937+0j)
>>> num + num.conjugate() # num.real * 2
(88+0j)
```

Booleani

In questa sezione iniziamo a parlare del meccanismo di *ereditarietà* tra i tipi, che vedremo in dettaglio nella seconda parte del libro, dedicata alla programmazione a oggetti.

Introduzione al concetto di ereditarietà tra i tipi

Nei linguaggi che supportano il paradigma di programmazione a oggetti esiste una sintassi per far ereditare a una classe gli attributi di un'altra classe. Quando una classe *B* eredita da una classe *A* si dice che *B* *deriva* da *A* o anche che *B* è una *sottoclasse* (in inglese *subclass*) di *A*. Data una classe *A* che ha un attributo `foo`:

```
>>> class A:
...     foo = 'python'
```

per definire una classe *B* che eredita da *A* si pone quest'ultima tra parentesi tonde nella definizione di *B*:

```
>>> class B(A):
...     pass
```

Come possiamo vedere, il tipo `B` e le sue istanze ereditano gli attributi di `A`:

```
>>> B.foo
'python'
>>> b = B() # Creiamo una istanza di 'B'
>>> b.foo
'python'
```

La funzione built-in `issubclass()` restituisce `True` quando la classe che le viene passata come primo argomento è una sottoclasse di quella passata come secondo argomento:

```
>>> issubclass(B, A)
True
```

Vediamo di capire con un esempio perché è utile l'ereditarietà. Supponiamo di aver definito una classe `Person` che rappresenta una generica persona, e che questa abbia gli attributi `nome` e `cognome`. Se volessimo definire una classe che rappresenta un professore, potremmo creare una classe `Professor` che deriva dalla classe `Person`. In questo modo la classe `Professor` eredita da `Person` gli attributi `nome` e `cognome`, e in più può specializzarsi definendo ulteriori attributi tipici di un professore.

Il legame tra i tipi `bool` e `int`

Il tipo `bool` fa parte della famiglia dei tipi numerici in quanto derivato dal tipo `int`:

```
>>> issubclass(bool, int) # La classe 'bool' è sottoclasse di 'int'
True
```

Il tipo `bool` ha solo due istanze. Una ha il significato di *vero*, ed è rappresentata dalla parola chiave `True`, mentre l'altra ha il significato di *falso* ed è rappresentata dalla parola chiave `False`:

```
>>> type(True), type(False)
(<class 'bool'>, <class 'bool'>)
```

Come abbiamo detto nel [Capitolo 1](#), `True` e `False` nelle operazioni matematiche vengono valutate come gli interi `1` e `0`:

```
>>> True + 9, False + 9, True / 1, False / 1
(10, 9, 1.0, 0.0)
```

Gli oggetti che appartengono alla categoria delle sequenze e quelli che appartengono alla categoria delle mappature sono considerati falsi se vuoti, altrimenti veri:

```
>>> bool([], bool([0, 0, 0, 0])
(False, True)
>>> bool(), bool((0, 0, 0, 0))
(False, True)
>>> bool({}), bool({'': 0})
(False, True)
>>> bool(""), bool(' ')
(False, True)
>>> bool(set()), bool({0}) # `set()` restituisce un set vuoto
(False, True)
```

Ogni oggetto numerico è considerato *falso* se il suo valore è zero, altrimenti è considerato *vero*:

```
>>> bool(0), bool(10), bool(-1), bool(-0)
(False, True, True, False)
>>> bool(0.0), bool(-0.0001), bool(-0.0)
(False, True, False)
>>> bool(1j), bool(0j)
(True, False)
```

Espressioni logiche

Le parole chiave `or` e `and` consentono di effettuare un *or* e un *and* logici. Una operazione di *or* tra oggetti è valutata `True` se almeno un oggetto è considerato vero:

```
>>> bool() or [] or "" or 0
False
>>> bool() or [] or "" or 0.1
True
```

Una operazione di *or* logico restituisce il primo oggetto valutato vero:

```
>>> () or [] or "" or 0.1 or 1 # Il numero `0.1` è il primo oggetto valutato `True`
0.1
```

Se nessuno degli oggetti è vero, allora viene restituito l'ultimo oggetto dell'espressione:

```
>>> () or [] or "" or 0 or set()
set()
```

Una operazione di *and* tra oggetti è valutata `True` se tutti gli oggetti sono considerati veri:

```
>>> bool((1, 2) and [1] and 'python' and {1: 'uno'})
True
>>> bool((1, 2) and [1] and 'python' and {})
False
```

L'operazione restituisce il primo oggetto considerato `False`, oppure l'ultimo oggetto valutato se tutti sono considerati `True`:

```
>>> (1, 2) and [1] and " and {}
"
>>> (1, 2) and [1] and 'python' and {1: 'uno'}
{1: 'uno'}
```

Librerie di terze parti

Chi lavora in campo scientifico può avvalersi di diverse librerie di terze parti. Ad esempio, per il calcolo scientifico sono disponibili *NumPy* e *SciPy*; per creare dei grafici vi è l'ottima libreria *Matplotlib*, per l'astronomia vi è *Astropy*, come shell interattiva vi è *IPython* ecc. Inoltre, a partire da Python 3.4 è disponibile il modulo `statistics` della libreria standard che fornisce il supporto per le più comuni funzioni statistiche. Per maggiori informazioni su questo nuovo modulo, si può consultare la PEP-0450, oppure la documentazione ufficiale, alla pagina <http://docs.python.org/3/library/statistics.html>.

Nel Capitolo 4 vedremo come installare le librerie di terze parti.

Operazioni e funzioni built-in utilizzabili con gli oggetti iterabili

In questa sezione parleremo sia delle operazioni comuni agli oggetti iterabili, sia delle funzioni built-in che prendono o restituiscono oggetti iterabili. Più avanti in questo capitolo parleremo delle operazioni e delle funzioni built-in specifiche delle sequenze.

Operazioni comuni agli oggetti iterabili

Le principali operazioni comuni agli oggetti iterabili sono lo *spacchettamento*, l'*iterazione* e il *test di appartenenza*. Le abbiamo viste tutte e tre nel precedente capitolo; qui ci proponiamo sia di approfondirle che di riportarle all'interno di una stessa sezione dedicata agli oggetti iterabili.

Spacchettamento

Gli oggetti iterabili supportano lo *spacchettamento*:

```
>>> a, b, c = (1, 2, 3) # Le parentesi non sono necessarie
>>> a
1
>>> a, b, c = 'foo'
>>> a
'f'
>>> open('myfile').read() # File contenente tre linee
'prima linea\nseconda linea\nterza linea\n'
>>> a, b, c = open('myfile') # Il file viene spacchettato linea per linea
>>> b
'seconda linea\n'
```

È importante tenere a mente che il dizionario non ha memoria dell'ordine di inserimento delle chiavi:

```
>>> a, b, c = {'name': 'Mario', 'surname': 'Rossi', 'age': 22} # Spacchettamento casuale
>>> a
'age'
```

Affinché si abbia uno spacchettamento, le etichette alla sinistra dell'assegnamento devono essere contenute in una tupla o in una lista. Quando vi è una sola etichetta, bisogna utilizzare una sintassi esplicita, racchiudendo l'etichetta in una lista oppure utilizzando la virgola per indicare

la presenza di una tupla:

```
>>> [a] = ['python']
>>> a
'python'
>>> a, = ['python']
>>> a
'python'
```

Esiste un'altra forma di spaccettamento, chiamata *spacchettamento esteso* (in inglese *extended iterable unpacking*, vedi PEP-3132), che si differenzia dallo spaccettamento classico per la presenza di una etichetta contrassegnata da un asterisco. A quest'ultima viene assegnata una lista contenente una parte degli elementi dell'oggetto iterabile:

```
>>> *a, b, c = 'python'
>>> a
['p', 'y', 't', 'h']
>>> b
'o'
>>> c
'n'
```

Come possiamo vedere, gli ultimi due elementi della stringa sono stati assegnati alle etichette b e c, e i rimanenti elementi sono stati inseriti in una lista assegnata ad a. L'etichetta con l'asterisco può trovarsi in una posizione qualsiasi:

```
>>> a, *b, c = 'python'
>>> a
'p'
>>> b
['y', 't', 'h', 'o']
>>> c
'n'
>>> a, b, *c = 'python'
>>> a
'p'
>>> b
'y'
>>> c
['t', 'h', 'o', 'n']
>>> *a, = 'python'
>>> a
['p', 'y', 't', 'h', 'o', 'n']
```

Vediamo cosa accade nello spaccettamento esteso quando il numero complessivo di etichette sulla sinistra dell'operatore di assegnamento è uguale

o maggiore di uno rispetto al numero degli elementi dell'oggetto iterabile:

```
>>> *a, b, c = [1, 2, 3]
>>> a, b, c
([1], 2, 3)
>>> *a, b, c, d = [1, 2, 3]
>>> a, b, c, d
([], 1, 2, 3)
>>> a, *b, c, d = [1, 2, 3]
>>> a, b, c, d
(1, [], 2, 3)
```

Test di appartenenza

La parola chiave `in` consente di verificare l'appartenenza di un elemento a un oggetto iterabile:

```
>>> 'yth' in 'python'
True
>>> (1, 2) in ['a', 'b', (1, 2)]
True
>>> 'name' in {'age': 33, 'name': 'Beppe'} # 'name' appartiene alle chiavi?
True
>>> 'Beppe' in {'age': 33, 'name': 'Beppe'} # 'Beppe' appartiene alle chiavi?
False
>>> 'prima\n' in open('myfile') # La stringa 'prima\n' appartiene alle linee del file?
True
```

La parola chiave `not` può essere usata in combinazione con `in` per verificare la *non* appartenenza:

```
>>> 'ythz' not in 'python'
True
>>> (1, 2) not in ['a', 'b', (1, 2)]
False
>>> 'Beppe' not in {'age': 33, 'name': 'Beppe'} # Non appartiene alle chiavi?
True
>>> 'prima' not in open('myfile') # La stringa 'prima' non appartiene alle linee del file?
True
```

Iterazione

Come sappiamo, la principale caratteristica degli oggetti iterabili è che si può accedere ai loro elementi, uno per volta, tramite l'istruzione `for`:

```
>>> for line in open('myfile'):
...     print(line, end=")
...
prima linea
seconda linea
```

terza linea

Se gli elementi di un oggetto iterabile sono, a loro volta, degli oggetti iterabili, allora in un contesto di iterazione è possibile spaccettarli:

```
>>> for *i, j in [open('myfile'), ('a', 'b'), 'python']:  
...     print(i, j)  
...  
['prima linea\n', 'seconda linea\n'] terza linea  
['a'] b  
['p', 'y', 't', 'h', 'o'] n
```

Funzioni built-in che prendono o restituiscono oggetti iterabili

Dato un oggetto iterabile che indichiamo con `iterable`, la chiamata `all(iterable)` restituisce `True` se per ogni `x` appartenente a `iterable` la chiamata `bool(x)` restituisce `True`, o se l'oggetto iterabile è vuoto:

```
>>> all(['a', (1, 2), {'uno': 1, 'due': 2}])  
True  
>>> all([0, 0, 0, 0])  
False  
>>> all({0: 'zero', 1: 'uno'}) # Restituisce 'False' perché 'bool(0)' restituisce 'False'  
False  
>>> all({0: 'zero', 1: 'uno'}) # Restituisce 'True' perché tutte le chiavi sono 'True'  
True  
>>> any([]) # Restituisce True se l'oggetto iterabile e' vuoto
```

La funzione built-in `any()` restituisce `True` se vi è almeno un elemento `x` dell'oggetto iterabile passato come argomento per cui `bool(x)` restituisce `True`:

```
>>> any([0, 1]) # Restituisce 'True' se 'bool(x)' è 'True' per almeno un 'x' dell'oggetto iterabile  
True  
>>> any([0, 0, 0, 0])  
False  
>>> any([]) # Restituisce False se l'oggetto iterabile e' vuoto  
False
```

Le funzioni `max()`, `min()` restituiscono l'elemento massimo e minimo di un oggetto iterabile:

```
>>> max([7, 9, 10, 3]) # Argomento iterabile  
10  
>>> max('a', 'c', 'b') # Numero arbitrario di argomenti confrontabili  
'c'
```

Queste accettano un argomento opzionale `key` che consente di applicare una

regola agli elementi dell'oggetto iterabile. Ad esempio, per ottenere gli elementi dell'oggetto iterabile aventi lunghezza massima e minima possiamo applicare la chiave `len()` alle funzioni `max()` e `min()`:

```
>>> max([(1, 2, 3), 'python', {'quattro': 4, 'cinque': 5, 'sei': 6, 'otto': 8}], key=len)
'python'
>>> min([(1, 2, 3), 'python', {'quattro': 4, 'cinque': 5, 'sei': 6, 'otto': 8}], key=len)
(1, 2, 3)
```

Se l'oggetto iterabile non ha elementi, entrambe le funzioni sollevano una eccezione:

```
>>> max([])
Traceback (most recent call last):
...
ValueError: max() arg is an empty sequence
>>> min([])
Traceback (most recent call last):
...
ValueError: min() arg is an empty sequence
```

A partire da Python 3.4 `min()` e `max()` accettano un secondo argomento opzionale, chiamato `default`, che serve per indicare l'oggetto da restituire nel caso in cui l'oggetto iterabile sia vuoto:

```
>>> max([], default=0)
0
>>> min([], default=0)
0
```

Per calcolare il massimo e il minimo in valore assoluto tra una serie di numeri interi, possiamo utilizzare la funzione built-in `abs()` come chiave:

```
>>> max([100, -3, 44, -150], key=abs)
-150
>>> min([100, -3, 44, -150], key=abs)
-3
```

La funzione `sum()` calcola la somma degli elementi di un oggetto iterabile:

```
>>> sum([1, 2, 3, 4])
10
>>> sum(range(10))
45
>>> sum({1: 'uno', 2: 'due'})
3
```

Accetta un argomento opzionale (per default vale 0) che consente di indicare un *offset* da aggiungere alla somma:

```
>>> sum([1, 2, 3, 4], 5) # Aggiunge `5` alla somma
15
```

NOTA

In realtà `sum()` non somma solamente dei numeri. Può, ad esempio, sommare liste o tuple:

```
>>> sum(['b', 'c'], [4], ['a'])
['a', 'b', 'c', 4]
>>> sum((('b', 'c'), (4)), ('a',))
('a', 'b', 'c', 4)
```

Nonostante vi siano anche queste possibilità di utilizzo, consigliamo di utilizzare `sum()` solamente per effettuare la somma di sequenze di numeri, principalmente per motivi di efficienza, come vedremo nella sezione *Il modulo itertools della libreria standard*.

Inoltre dobbiamo tenere presente che in nessun caso è possibile concatenare le stringhe:

```
>>> sum('abc', '-')
Traceback (most recent call last):
...
TypeError: sum() can't sum strings [use ''.join(seq) instead]
>>> sum(['a', 'b', 'c', '-'])
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Come suggerisce il messaggio di errore, il modo migliore per concatenare le stringhe è quello di usare il metodo `str.join()`:

```
>>> '-'.join('ciao')
'c-i-a-o'
>>> '-'.join(['c', 'i', 'a', 'o'])
'c-i-a-o'
```

La funzione `sorted()` restituisce una lista contenente gli elementi dell'oggetto iterabile disposti in modo ordinato:

```
>>> sorted((55, 33, 11, 22))
[11, 22, 33, 55]
```

Possiamo utilizzare la funzione `sorted()` per iterare sulle chiavi di un dizionario in modo ordinato, dalla più piccola alla più grande:

```
>>> d = {11: 'undici', 22: 'ventidue', 33: 'trentatre'}
>>> for k in d: # Itero sulle chiavi in modo casuale
... print(k, d[k], sep=' -> ')
...
33 -> trentatre
11 -> undici
22 -> ventidue
>>> for k in sorted(d): # Itero sulle chiavi, dalla più piccola alla più grande
... print(k, d[k], sep=' -> ')
...
11 -> undici
22 -> ventidue
33 -> trentatre
```

Come per le funzioni `max()` e `min()`, anche la funzione `sorted()` accetta l'argomento opzionale `key`:

```
>>> sorted([(1, 2, 3), 'python', {'quattro': 4, 'cinque': 5, 'sei': 6, 'otto': 8}])
Traceback (most recent call last):
...
TypeError: unorderable types: str() < tuple()
>>> sorted([(1, 2, 3), 'python', {'quattro': 4, 'cinque': 5, 'sei': 6, 'otto': 8}], key=len)
[(1, 2, 3), {'sei': 6, 'cinque': 5, 'otto': 8, 'quattro': 4}, 'python']
```

Ha, inoltre, un terzo argomento opzionale, chiamato `reverse`, che consente di ordinare gli elementi al contrario:

```
>>> sorted([(1, 2, 3), 'python', {'quattro': 4, 'cinque': 5, 'sei': 6, 'otto': 8}],
... key=len, reverse=True)
['python', {'cinque': 5, 'quattro': 4, 'otto': 8, 'sei': 6}, (1, 2, 3)]
['python', {'sei': 6, 'cinque': 5, 'otto': 8, 'quattro': 4}, (1, 2, 3)]
```

NOTA

In Python 2 la funzione built-in `sorted()` può ordinare oggetti contenenti sia numeri sia stringhe:

```
>>> myseq = (1, 2, 'a', 'b') # Python 2.7
>>> sorted(myseq)
[1, 2, 'a', 'b']
```

Questo è un punto di rottura con Python 3:

```
>>> myseq = (1, 2, 'a', 'b') # Python 3
>>> sorted(myseq)
Traceback (most recent call last):
...
TypeError: unorderable types: str() < int()
```

La funzione `zip()` prende come argomenti un numero arbitrario di oggetti iterabili e restituisce un iteratore:

```
>>> z = zip([1, 2, 3], open('myfile'), 'abc')
>>> type(z)
<class 'zip'>
>>> hasattr(z, '__iter__'), hasattr(z, '__next__')
(True, True)
>>> z is z.__iter__()
True
```

Il metodo `z.__next__()` restituisce una tupla il cui elemento *i*-esimo è composto dagli elementi *i*-esimi degli oggetti iterabili passati come argomento:

```
>>> next(z)
(1, 'prima linea\n', 'a')
>>> next(z)
(2, 'seconda linea\n', 'b')
>>> next(z)
(3, 'terza linea\n', 'c')
```

Il numero di elementi che può restituire `iteratore.__next__()` è pari al numero di elementi dell'oggetto iterabile più piccolo passato come argomento:

```
>>> for item in zip(['a', 'b', 'c'], ([], ())):
...     print(item)
...
('a', [])
('b', ())
```

La funzione `map()` prende come argomenti una funzione e uno o più oggetti iterabili, e restituisce un iteratore:

```
>>> m = map(bin, (1, 2))
>>> type(m)
<class 'map'>
>>> hasattr(m, '__iter__'), hasattr(m, '__next__')
(True, True)
>>> m is m.__iter__()
True
```

True

L'iteratore a ogni iterazione passa gli elementi degli oggetti iterabili come argomenti della funzione. Consideriamo, ad esempio, il caso di un solo oggetto iterabile:

```
>>> for i in map(bin, [2, 3, 4]):
...     print(i, end=' ')
...
0b10 0b11 0b100
```

Come possiamo vedere, all'etichetta `i` sono stati assegnati, rispettivamente, `bin(2)`, `bin(3)` e `bin(4)`.

Se si passa a `map()` più di un oggetto iterabile, allora a ogni iterazione da ogni oggetto iterabile viene preso un elemento, e tutti questi vengono passati come argomenti alla funzione. Il numero di iterazioni è pari al numero di elementi dell'oggetto iterabile più piccolo. Ad esempio, nel seguente caso:

```
>>> def product(x, y):
...     return x * y
...
>>> for i in map(product, (2, 3, 4), (10, 20)):
...     print(i)
...
20
60
```

l'etichetta `i` alla prima iterazione viene assegnata al risultato di `product(2, 10)`, mentre alla seconda viene assegnata al risultato di `product(3, 20)`.

La funzione `filter(fun or None, iterable)` prende come primo argomento o una funzione oppure `None`, e come secondo argomento un oggetto iterabile. L'oggetto restituito da `filter()` è un iteratore:

```
>>> f = filter(None, [1, 2])
>>> type(f)
<class 'filter'>
>>> hasattr(f, '__iter__'), hasattr(f, '__next__')
(True, True)
>>> f is f.__iter__()
True
```

Quando il primo argomento di `filter()` è una funzione, diciamo `fun()`, allora `filter()` chiama `fun()` passandole uno per uno gli elementi `item` dell'oggetto iterabile `iterable`. L'iteratore restituito da `filter()` contiene gli elementi `item` per i quali `fun(item)`

restituisce `True`. Nel seguente esempio l'unico risultato di `foo(item)` valutato `True` è 4, ottenuto quando `item` è pari a 2, per cui l'iteratore restituito da `filter()` contiene solo l'elemento 2:

```
>>> def foo(x):
...     return x * x * (x - 1)
...
>>> foo(0), foo(1), foo(2)
(0, 0, 4)
>>> for i in filter(foo, range(3)):
...     print(i)
...
2
```

Se, come primo argomento, viene passato `None`, allora `filter()` restituisce gli elementi dell'oggetto iterabile che sono valutati `True`:

```
>>> for i in filter(None, range(3)):
...     print(i)
...
1
2
```

Vi sono, infine, le classi `range` ed `enumerate`, delle quali abbiamo discusso in modo dettagliato nel [Capitolo 1](#).

Il modulo `itertools` della libreria standard

Come possiamo immaginare, il modulo `itertools` fornisce gli strumenti per lavorare in modo semplice con diversi tipi di *iteratori*, che si rivelano molto utili in vari ambiti. Vedremo qualche esempio, giusto per capire di cosa si tratta.

La classe `itertools.count` restituisce un iteratore che consente di effettuare conteggi a partire da un certo numero iniziale:

```
>>> import itertools
>>> counter = itertools.count(100)
>>> next(counter)
100
>>> next(counter)
101
>>> next(counter)
102
```

È possibile specificare anche lo *step*:

```
>>> counter = itertools.count(100, 2)
>>> next(counter)
100
>>> next(counter)
102
>>> next(counter)
104
```

La classe `itertools.cycle` consente di effettuare dei cicli:

```
>>> cycle = itertools.cycle([10, 20, 30])
>>> next(cycle) # Primo elemento
10
>>> next(cycle) # Secondo elemento
20
>>> next(cycle) # Ultimo elemento
30
>>> next(cycle) # Riprende dal primo elemento
10
```

La classe `itertools.combinations` consente di iterare sulle *combinazioni* degli elementi di un oggetto iterabile:

```
>>> list(itertools.combinations('abcd', 2))
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'd')]
>>> list(itertools.combinations('abcd', 3))
[('a', 'b', 'c'), ('a', 'b', 'd'), ('a', 'c', 'd'), ('b', 'c', 'd')]
```

La classe `itertools.permutations` consente di iterare sulle *permutazioni* degli elementi di un oggetto iterabile:

```
>>> list(itertools.permutations('abc'))
[('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'), ('b', 'c', 'a'), ('c', 'a', 'b'), ('c', 'b', 'a')]
>>> list(itertools.permutations('abc', 2))
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]
```

La classe `itertools.chain` prende come argomenti degli oggetti iterabili e restituisce un iteratore che itera su tali argomenti:

```
>>> for item in itertools.chain([1, 2, 3], 'abc', (11, 12, 13)):
...     print(item)
...
1
2
3
a
b
c
11
```

```

12
13
>>> list(itertools.chain([1, 2, 3], 'abc', (11, 12, 13)))
[1, 2, 3, 'a', 'b', 'c', 11, 12, 13]

```

Se, invece, vogliamo concatenare degli oggetti iterabili che risultano elementi di un oggetto iterabile, allora il modo migliore per farlo è quello di utilizzare la funzione `itertools.chain.from_iterable()`:

```

>>> for item in itertools.chain.from_iterable([[1, 2, 3], 'abc', {'uno': 1, 'due': 2}]):
...     print(item)
...
1
2
3
a
b
c
uno
due
>>> list(itertools.chain.from_iterable([[1, 2, 3], 'abc', {'uno': 1, 'due': 2}]))
[1, 2, 3, 'a', 'b', 'c', 'uno', 'due']

```

NOTA

Per motivi di efficienza è preferibile usare la funzione `itertools.chain.from_iterable()` piuttosto che la funzione built-in `sum()`:

```

$ python -m timeit -s "lists=[[i]*i for i in range(100)]" \
> "import itertools; list(itertools.chain.from_iterable(lists))"
10000 loops, best of 3: 47.7 usec per loop
$ python -m timeit -s "lists=[[i]*i for i in range(100)]" "sum(lists, [])"
1000 loops, best of 3: 547 usec per loop

```

Vi sono anche altri metodi alternativi a `itertools.chain.from_iterable()`, che vedremo meglio quando parleremo delle liste:

```

$ python -m timeit -s "flat=[]" "lists=[[i]*i for i in range(100)]" \
> "for lst in lists: flat += lst"
10000 loops, best of 3: 53.1 usec per loop
$ python -m timeit -s "flat=[]" "lists=[[i]*i for i in range(100)]" \
> "for lst in lists: flat.extend(lst)"
10000 loops, best of 3: 55.7 usec per loop

```

Questa è stata solo una brevissima introduzione al modulo `itertools`. Per maggiori informazioni rimandiamo alla documentazione ufficiale:

<http://docs.python.org/3/library/itertools.html>.

Gli insiemi matematici

I tipi built-in `set` e `frozenset` rappresentano nel mondo Python gli *insiemi matematici*. La differenza tra i due tipi è che le istanze della classe `set` sono oggetti mutabili che possono essere creati tramite dei letterali, mentre quelle della classe `frozenset` sono immutabili e non hanno associato alcun letterale.

Set

Nel capitolo precedente abbiamo detto che le istanze del tipo `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, separati con una virgola:

```
>>> s1 = set([1, 'ciao', 2, ('a', 'b')]) # Creazione di un set tramite la classe `set`
>>> s2 = {1, 'py', ('a', 'b'), 3} # Creazione di un set tramite letterale
```

Il letterale `{}` rappresenta un dizionario vuoto, per cui per creare un set vuoto è necessario chiamare la classe `set` senza argomenti:

```
>>> type(set()), type({}) # Per creare un set vuoto si usa la classe `set`
(<class 'set'>, <class 'dict'>)
```

Il set è un oggetto mutabile:

```
>>> s2.add(4) # Aggiungo un elemento al set
>>> s2
{('a', 'b'), 1, 'py', 3, 4}
```

ma i suoi elementi non possono essere oggetti mutabili:

```
>>> s2.add([1, 2]) # Gli elementi di un set devono essere oggetti immutabili
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

Gli elementi all'interno di un set sono *unici*, ovvero possono comparire solo una volta:

```
>>> s2.add('py') # Un set non può contenere duplicati
>>> s2
{'a', 'b'}, 1, 'py', 3, 4}
```

I set, inoltre, sono oggetti iterabili, ma, come abbiamo detto, non sono ordinati, nel senso che non hanno memoria dell'ordine di inserimento degli elementi:

```
>>> for item in {'a', 'b', (1, 2)}:
...     print(item)
...
(1, 2)
a
b
```

La funzione built-in `len()` ci consente di ottenere il numero di elementi di un set:

```
>>> len({1, 2, 3})
3
```

Operazioni tra set

Gli operatori fanno sì che i set si comportino esattamente come gli insiemi matematici.

L'operazione di *intersezione* restituisce un set contenente gli elementi comuni ai suoi operandi:

```
>>> {'a', 'b', 'c', 'd'} & {'a', 'c', 'e'}
{'a', 'c'}
```

L'*unione* restituisce un set contenente tutti gli elementi dei due operandi:

```
>>> {'a', 'b', 'c', 'd'} | {'a', 'c', 'e'}
{'d', 'e', 'a', 'b', 'c'}
```

La *differenza* restituisce un set contenente gli elementi del primo set che non sono presenti nel secondo:

```
>>> {'a', 'b', 'c', 'd'} - {'a', 'c', 'e'}
{'d', 'b'}
```

La *differenza simmetrica* tra due set restituisce un set contenente gli elementi che sono presenti in uno dei due set ma non nell'altro. Ad esempio, nel seguente caso:

```
>>> {'a', 'b', 'c', 'd'} ^ {'a', 'c', 'e'}
{'d', 'e', 'b'}
```

gli elementi *b* e *d* sono presenti nel primo set ma non nel secondo, per cui fanno parte del risultato, così come l'elemento *e*, poiché fa parte del secondo set ma non del primo. La differenza simmetrica tra due set s_1 e s_2 è quindi equivalente sia a $(s_1 - s_2) \cup (s_2 - s_1)$ sia a $(s_1 \cup s_2) - (s_1 \cap s_2)$:

```
>>> ({'a', 'b', 'c', 'd'} - {'a', 'c', 'e'}) | ({'a', 'c', 'e'} - {'a', 'b', 'c', 'd'})
{'d', 'e', 'b'}
>>> ({'a', 'b', 'c', 'd'} | {'a', 'c', 'e'}) - ({'a', 'b', 'c', 'd'} & {'a', 'c', 'e'})
{'d', 'e', 'b'}
```

Gli operatori $>$, \supseteq , $<$ e \subseteq consentono di verificare l'appartenenza di un set a un altro set:

```
>>> {1, 2, 3} > {1, 2} # Il primo insieme include in senso stretto il secondo?
True
>>> {1, 2} > {1, 2}, {1, 2} >= {1, 2} # Inclusione in senso stretto (>) e in senso largo (>=)
(False, True)
>>> {1, 2} < {1, 2, 3} # Il primo insieme è incluso nel secondo?
True
>>> {1, 2} <= {1, 2}, {1, 2} < {1, 2}
(True, False)
```

In tutte le operazioni entrambi gli operandi devono essere di tipo set:

```
>>> {1, 2} & (1, 2, 3)
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for &: 'set' and 'tuple'
```

I metodi dei set

I set hanno dei metodi che consentono di svolgere le operazioni che abbiamo appena visto:

```
>>> s1 = {1, 2, 3, 4}
>>> s2 = {3, 4, 5}
>>> s1.intersection(s2) # Equivale a `s1 & s2`
{3, 4}
>>> s1.union(s2) # Equivale a `s1 | s2`
{1, 2, 3, 4, 5}
>>> s1.difference(s2) # Equivale a `s1 - s2`
{1, 2}
>>> s1.symmetric_difference(s2) # Equivale a `s1 ^ s2`
{1, 2, 5}
```

I metodi sono, però, più generici dei corrispondenti operatori, poiché consentono di effettuare operazioni non solo tra oggetti di tipo `set`, ma anche tra `set` e oggetti iterabili:

```
>>> open('foo', 'w').write('first line\nsecond line') # Creiamo un file di due linee
22
>>> s = {1, 2, 3, 4}
>>> s.union(open('foo'))
{1, 2, 3, 4, 'first line\n', 'second line'}
>>> s.union(['python', 5, 6, ('a', 'b')])
{1, 2, 3, 4, 5, 6, 'python', ('a', 'b')}
>>> s.union('python')
{1, 2, 3, 4, 'p', 't', 'y', 'n', 'o', 'h'}
>>> s.intersection({1: 'uno', 3: 'tre'}) # L'iteratore di un dizionario itera sulle chiavi
{1, 3}
>>> s & {1: 'uno', 3: 'tre'} # Non è possibile fare la stessa cosa con gli operatori
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for &: 'set' and 'dict'
```

I metodi appena visti non modificano i `set`, ma ne restituiscono di nuovi:

```
>>> s # Le precedenti chiamate ai metodi non hanno modificato il set `s`
{1, 2, 3, 4}
```

Esistono, però, dei metodi che eseguono l'operazione modificando il `set`, senza restituire alcun valore:

```
>>> s.difference_update([3, 4]) # Si osservi che non restituisce alcun oggetto
>>> s # Il set è stato modificato
{1, 2}
>>> s.symmetric_difference_update((1, 3, 4))
>>> s
{2, 3, 4}
>>> s.intersection_update({2: 'due'})
>>> s
{2}
```

Il metodo `set.union_update()` non esiste perché `set.update()` si comporta esattamente come una *union update*, e, come sappiamo, lo zen dice che *Ci dovrebbe essere un modo ovvio -- e preferibilmente solo uno -- di fare le cose*:

```
>>> s.update([1, 2, 3, 4])
>>> s
{1, 2, 3, 4}
```

Vi sono anche altri metodi che consentono di modificare un `set`. I metodi `set.discard()`, `set.remove()` e `set.pop()` rimuovono un elemento dal `set`. Il primo prende

come argomento l'elemento da rimuovere e non restituisce alcun valore:

```
>>> s.discard(2) # Rimuove l'elemento `2` dal set e non restituisce nulla
>>> s
{1, 3, 4}
```

Se l'elemento non esiste, `set.discard()` non solleva alcuna eccezione:

```
>>> s.discard(2) # Se l'elemento non esiste non succede nulla
>>> s
{1, 3, 4}
```

Il metodo `set.remove()`, a differenza di `set.discard()`, solleva una eccezione se l'elemento non esiste:

```
>>> s.remove(1) # Rimuove l'elemento
>>> s
{3, 4}
>>> s.remove(1) # L'elemento da rimuovere non esiste e lo segnala sollevando una eccezione
Traceback (most recent call last):
...
KeyError: 1
```

Il metodo `s.pop()` restituisce un elemento (in modo non deterministico) e lo rimuove dal set:

```
>>> s = {'a', 'b'}
>>> s.pop() # Restituisce un elemento, in modo non deterministico
'b'
>>> s
{'a'}
>>> s.pop()
'a'
>>> s
set()
```

Se il set è vuoto solleva una eccezione:

```
>>> s.pop() # Il set è vuoto e lo segnala sollevando una eccezione
Traceback (most recent call last):
...
KeyError: 'pop from an empty set'
```

Set comprehension

È possibile creare un `set` in modo conciso utilizzando una sintassi chiamata *set comprehension*, che, come possiamo immaginare, è analoga a quella vista per

le *list comprehension*:

```
>>> {bin(x) for x in range(5)}  
{'0b100', '0b1', '0b0', '0b11', '0b10'}
```

Si può ottenere lo stesso risultato utilizzando il metodo `set.add()`:

```
>>> s = set()  
>>> for x in range(5):  
... s.add(bin(x))  
...  
>>> s  
{'0b1', '0b0', '0b100', '0b11', '0b10'}
```

La *set comprehension* non solo ha una sintassi più concisa, ma viene anche eseguita in modo più veloce del codice equivalente che fa uso del metodo `set.add()`:

```
$ python -m timeit "{i for i in range(100)}"  
100000 loops, best of 3: 7.63 usec per loop  
$ python -m timeit "s = set()" "for i in range(100): s.add(i)"  
100000 loops, best of 3: 14 usec per loop
```

NOTA

Il modulo `timeit` consente di ottenere in modo semplice una misura del tempo impiegato per l'esecuzione di piccole porzioni di codice. Se vogliamo approfondire l'argomento, possiamo consultare la documentazione online alla pagina <http://docs.python.org/3/library/timeit.html>.

All'interno della *set comprehension* è possibile utilizzare la parola chiave `if`. Ad esempio, il seguente *set* contiene i numeri pari compresi tra 0 e 10 (escluso):

```
>>> s = set()  
>>> for i in range(10):  
... if i % 2 == 0:  
... s.add(i)  
...  
>>> s  
{0, 8, 2, 4, 6}
```

La corrispondente *set comprehension* è la seguente:

```
>>> {i for i in range(10) if i % 2 == 0}
{0, 2, 4, 6, 8}
```

È possibile anche annidare più cicli `for`:

```
>>> {(x, y) for x in [1, 2, 3] for y in [2, 3, 4] if x != y}
{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}
```

Il ciclo più esterno itera sulla prima lista `[1, 2, 3]`. A ogni iterazione (quindi, per ogni elemento della prima lista) il ciclo più interno itera sulla seconda lista `[2, 3, 4]`. Per ogni coppia `x` e `y` così formata, se risulta che `x != y` la tupla `(x, y)` viene aggiunta al set. Il codice equivalente è il seguente:

```
>>> s = set()
>>> for x in [1, 2, 3]:
...     for y in [2, 3, 4]:
...         if x != y:
...             s.add((x, y))
...
>>> s
{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}
```

I set immutabili

Abbiamo detto che i set sono oggetti mutabili che possono avere come elementi solo oggetti immutabili. Questo significa che un set non può contenere un altro set:

```
>>> {1, 2, {1, 2}}
Traceback (most recent call last):
...
TypeError: unhashable type: 'set'
```

Esiste, però, un tipo chiamato `frozenset` le cui istanze sono equivalenti a set immutabili, che consentono di creare insiemi contenenti oggetti mutabili:

```
>>> f = frozenset([1, 2]) # Non esistono letterali per i frozenset
>>> f
frozenset({1, 2})
>>> type(f)
<class 'frozenset'>
>>> {1, 2, f} # Un set che contiene un frozenset
{frozenset({1, 2}), 1, 2}
```

Poiché le istanze della classe `frozenset` sono oggetti immutabili, non hanno i

metodi che consentono di modificarle:

```
>>> set(dir(set)) - set(dir(frozenset))
{'remove', 'update', '__iand__', 'add', 'discard', 'clear', '__ior__', 'difference_update', '__isub__',
'symmetric_difference_update', 'pop', '__ixor__', 'intersection_update'}
```

È possibile effettuare delle operazioni tra `set` e `frozenset`, e il risultato è un `frozenset`:

```
>>> fs | {2, 3, 4}
frozenset({1, 2, 3, 4})
>>> fs & {2, 3, 4}
frozenset({2})
```

Dizionari

Come ormai sappiamo, i *dizionari* sono collezioni non ordinate di elementi rappresentati da coppie *chiave-valore*. Poiché il dizionario non ha memoria dell'ordine di inserimento degli elementi, i valori non hanno associato alcun indice e quindi la ricerca avviene per chiave:

```
>>> d = {'nome': 'Anna', 'professione': ['studentessa', 'lavoratrice'], 'web': 'www.foohome.it'}
>>> d['web'] # Accesso agli elementi per chiave
'www.foohome.it'
>>> len(d) # Restituisce il numero di elementi, ovvero il numero di coppie chiave-valore
3
```

I dizionari, quindi, sono oggetti che, data una chiave, restituiscono il corrispondente valore, ovvero consentono di ottenere una *mappatura* tra chiavi e valori. Per questo motivo, si dice che i dizionari appartengono alla categoria delle *mappature*. Mentre i valori possono essere oggetti di tipo arbitrario, le chiavi devono essere oggetti immutabili:

```
>>> d = {[1, 2]: 'uno e due'} # Le liste sono oggetti mutabili
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

Un dizionario è un oggetto mutabile:

```
>>> d = {1: ['uno'], 2: ['due', 'two']}
>>> d[3] = ['tre', 'three']
>>> d
{1: ['uno'], 2: ['due', 'two'], 3: ['tre', 'three']}
```

L'istruzione `del d[key]` rimuove dal dizionario `d` la chiave `key` e il corrispondente valore:

```
>>> del d[3]
>>> d
{1: ['uno'], 2: ['due', 'two']}
```

I valori di un dizionario sono dei riferimenti ai corrispondenti oggetti:

```
>>> mylist = d[1]
```

```
>>> mylist
['uno']
>>> d[1].append('one')
>>> d
{1: ['uno', 'one'], 2: ['due', 'two']}
>>> mylist
['uno', 'one']
```

I dizionari possono essere creati anche mediante la classe `dict`:

```
>>> dict() # Dizionario vuoto
{}

```

È possibile passare alla classe `dict` un oggetto iterabile avente per elementi delle coppie di oggetti:

```
>>> dict([('colore', 'giallo'), ('numero', 33)]) # Lista di tuple
{'numero': 33, 'colore': 'giallo'}
```

Si può utilizzare anche la notazione `chiave=valore`:

```
>>> dict(colore='giallo', numero=33)
{'numero': 33, 'colore': 'giallo'}
```

La funzione built-in `len()` ci consente di ottenere il numero di elementi di un dizionario:

```
>>> len({1: 'uno', 2: 'due', 3: 'tre'})
3
```

Così come per le liste e per i set, è possibile creare un dizionario in modo conciso mediante una *dictionary comprehension*, la cui sintassi si differenzia da quella della set comprehension per via della presenza del simbolo dei due punti, che separa la chiave dal corrispondente valore:

```
>>> {k: bin(k**2) for k in range(5) if k % 2 == 0}
{0: '0b0', 2: '0b100', 4: '0b10000'}
```

Metodi dei dizionari

Ecco i principali metodi dei dizionari:

```
>>> [name for name in dir(dict) if not name.startswith('__')]
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

Il metodo `dict.copy()` effettua una copia *shallow* del dizionario, ovvero una copia i cui elementi sono riferimenti ai corrispondenti elementi del dizionario originale:

```
>>> d = {100: 100, 'numeri': [1, 2, 3], 'lettere': ['a', 'b', 'c']}
>>> c = d.copy()
>>> d is c
False
>>> d['numeri'] is c['numeri']
True
>>> d[100] is c[100]
True
>>> d['numeri'][2] = 'tre'
>>> d
{'numeri': [1, 2, 'tre'], 100: 100, 'lettere': ['a', 'b', 'c']}
>>> c
{'numeri': [1, 2, 'tre'], 100: 100, 'lettere': ['a', 'b', 'c']}
>>> d.clear() # Svuota il dizionario
>>> d, c
({}, {'numeri': [1, 2, 'tre'], 100: 100, 'lettere': ['a', 'b', 'c']})
```

Per effettuare una *copia profonda*, nella quale gli elementi della copia non sono riferimenti agli oggetti originali ma delle copie, si utilizza la funzione `deepcopy()` del modulo `copy`:

```
>>> d = {100: 100, 'numeri': [1, 2, 3], 'lettere': ['a', 'b', 'c']}
>>> import copy
>>> c = copy.deepcopy(d)
>>> d['numeri'] is c['numeri']
False
>>> d['numeri'][2] = 'tre'
>>> d
{'numeri': [1, 2, 'tre'], 100: 100, 'lettere': ['a', 'b', 'c']}
>>> c
{'numeri': [1, 2, 3], 100: 100, 'lettere': ['a', 'b', 'c']}
```

Il metodo `dict.fromkeys()` consente di creare un dizionario a partire da un oggetto iterabile i cui elementi sono immutabili:

```
>>> dict.fromkeys([1, 2, 3])
{1: None, 2: None, 3: None}
>>> {x: None for x in [1, 2, 3]} # Equivalente a `dict.fromkeys([1, 2, 3])`
{1: None, 2: None, 3: None}
>>> dict.fromkeys([1, 2, 3], 'foo') # Viene assegnato il valore 'foo' a tutte le chiavi
{1: 'foo', 2: 'foo', 3: 'foo'}
>>> {x: 'foo' for x in [1, 2, 3]} # Equivalente a `dict.fromkeys([1, 2, 3], 'foo')`
{1: 'foo', 2: 'foo', 3: 'foo'}
>>> f = open('myfile', 'w')
>>> f.writelines(['prima\n', 'seconda\n', 'terza\n'])
>>> f.close()
>>> dict.fromkeys(open('myfile'))
```

```
{'prima\n': None, 'seconda\n': None, 'terza\n': None}
```

Se si tenta di accedere a un elemento di un dizionario la cui chiave non esiste, viene sollevata una eccezione:

```
>>> d = dict(articolo='latte', quantità=5)
>>> d['marca']
Traceback (most recent call last):
...
KeyError: 'marca'
```

Se non si vuole gestire l'eccezione, si può fare un test di appartenenza:

```
>>> print(d['marca']) if 'marca' in d else print('marca` not in d')
`marca` not in d
```

oppure, in modo ancora più semplice e conciso, si può utilizzare il metodo `dict.get()`:

```
>>> d.get('articolo')
'latte'
>>> d.get('marca')
>>> d.get('marca', 'marca` not in d')
'marca` not in d'
```

Un altro metodo utile in questo contesto è `dict.setdefault(chiave[,valore])`. Questo, quando la chiave esiste, si comporta esattamente come `dict.get()`:

```
>>> d = {1: 'uno', 2: 'due'}
>>> d.setdefault(1)
'uno'
```

A differenza di `dict.get()`, se la chiave non è presente viene inserita nel dizionario e le viene assegnato per default `None` come valore:

```
>>> d.setdefault(3)
>>> d
{1: 'uno', 2: 'due', 3: None}
```

Se si vuole specificare il valore nel caso in cui la chiave non sia presente, lo si passa come secondo argomento:

```
>>> d.setdefault(4, 'quattro') # Il valore viene comunque restituito
'quattro'
>>> d
{1: 'uno', 2: 'due', 3: None, 4: 'quattro'}
```

```
>>> d.setdefault(4, 'otto')
'quattro'
>>> d
{1: 'uno', 2: 'due', 3: None, 4: 'quattro'}
```

Il metodo `dict.pop()` rimuove una data chiave dal dizionario e restituisce il corrispondente valore:

```
>>> d.pop(1)
'uno'
>>> d
{2: 'due', 3: None, 4: 'quattro'}
```

Se la chiave non esiste, solleva una eccezione:

```
>>> d.pop(1)
Traceback (most recent call last):
...
KeyError: 1
```

Si può evitare che venga sollevata una eccezione passando un secondo argomento. Questo viene restituito nel caso in cui la chiave non esista:

```
>>> d.pop(1, 'chiave non trovata')
'chiave non trovata'
```

Il metodo `dict.popitem()`, invece, rimuove e restituisce un tupla contenente una coppia chiave-valore arbitraria:

```
>>> d = {'name': 'Marco', 'age': 35}
>>> d.popitem()
('name', 'Marco')
>>> d.popitem()
('age', 35)
```

Se il dizionario è vuoto, solleva una eccezione:

```
>>> d.popitem()
Traceback (most recent call last):
...
KeyError: 'popitem(): dictionary is empty'
```

È possibile estendere e aggiornare un dizionario mediante il metodo `dict.update()`:

```
>>> d = dict(nome='Python', creatore='Guido')
>>> d
{'creatore': 'Guido', 'nome': 'Python'}
```

```
>>> d.update(dict(creatore='Guido van Rossum')) # Aggiorna il valore
>>> d
{'creatore': 'Guido van Rossum', 'nome': 'Python'}
>>> d.update({'voto': 10}) # Aggiorna inserendo una nuova coppia chiave-valore
>>> d
{'creatore': 'Guido van Rossum', 'voto': 10, 'nome': 'Python'}
```

Allo stesso modo della classe `dict`, il metodo `dict.update()` può prendere come argomento un oggetto iterabile i cui elementi sono coppie chiave-valore:

```
>>> d = {1: 'uno', 2: 'due'}
>>> d.update([(3, 'tre'), (4, 'quattro')])
>>> d
{1: 'uno', 2: 'due', 3: 'tre', 4: 'quattro'}
```

ed è anche possibile utilizzare la notazione `chiave=valore`:

```
>>> d.update(uno='one', due='two')
>>> d
{1: 'uno', 2: 'due', 3: 'tre', 4: 'quattro', 'uno': 'one', 'due': 'two'}
```

Dizionari e iterazione

Un dizionario è un oggetto iterabile e il suo iteratore itera sulle chiavi:

```
>>> d = {11: 'undici', 22: 'ventidue', 33: 'trentatre'}
>>> for k in d:
...     print(k)
...
33
11
22
```

Come abbiamo visto nella sezione *Funzioni built-in che prendono o restituiscono oggetti iterabili*, per iterare sulle chiavi di un dizionario in modo ordinato, dalla più piccola alla più grande o viceversa, si utilizza la funzione built-in `sorted()`:

```
>>> for k in sorted(d):
...     print(k, d[k], sep=' -> ')
...
11 -> undici
22 -> ventidue
33 -> trentatre
>>> for k in sorted(d, reverse=True):
...     print(k, d[k], sep=' -> ')
...
33 -> trentatre
```

22 -> ventidue
11 -> undici

È possibile iterare sulle chiavi, sui valori e sulle coppie chiave-valore in modo esplicito, mediante i metodi `dict.keys()`, `dict.values()` e `dict.items()`. Questi restituiscono degli oggetti iterabili detti *dictionary view object*:

```
>>> d = dict(nome='spaghetti', tipo='pasta', voto=10)
>>> type(d.keys()), type(d.values()), type(d.items())
(<class 'dict_keys'>, <class 'dict_values'>, <class 'dict_items'>)
>>> for k in d.keys():
...     print(k, end=' ')
...
voto nome tipo
>>> for v in d.values():
...     print(v, end=' ')
...
10 spaghetti pasta
>>> for item in d.items():
...     print(item, end=' ')
...
('voto', 10) ('nome', 'spaghetti') ('tipo', 'pasta')
```

Gli oggetti di tipo `dict_keys` supportano le operazioni dei set:

```
>>> d1 = dict(a=1, b=2, c=3)
>>> d2 = dict(a=2, d=4)
>>> d1.keys() | d2.keys()
{'d', 'c', 'b', 'a'}
>>> d1.keys() & d2.keys()
{'a'}
```

Anche i `dict_items` le supportano se i valori del dizionario sono oggetti immutabili:

```
>>> d1.items() | d1.keys()
{'c', 'b', 'a', ('a', 1), ('c', 3), ('b', 2)}
>>> d1.items() & {'a', '1'}
set()
>>> d1.items() & {'a', 1}
{('a', 1)}
```

Poiché i valori di un dizionario non sono unici, i `dict_value` non supportano le operazioni dei set.

Dizionari ordinati

Come sappiamo, i dizionari non sono oggetti ordinati, per cui non tengono memoria dell'ordine con cui inseriamo gli elementi:

```
>>> d = {}
>>> d[11] = 'undici'
>>> d[22] = 'ventidue'
>>> d[33] = 'trentatre'
>>> for k in d:
...     print(k, d[k], sep=' -> ')
...
33 -> trentatre
11 -> undici
22 -> ventidue
```

Se vogliamo che il dizionario abbia memoria dell'ordine di inserimento, dobbiamo creare un cosiddetto *dizionario ordinato* (*ordered dict*). I dizionari ordinati sono istanza della classe `OrderedDict`, definita nel modulo `collections`:

```
>>> from collections import OrderedDict
>>> d = OrderedDict()
>>> d[11] = 'undici'
>>> d[22] = 'ventidue'
>>> d[33] = 'trentatre'
>>> for k in d:
...     print(k, d[k], sep=' -> ')
...
11 -> undici
22 -> ventidue
33 -> trentatre
```

Se un elemento del dizionario viene aggiornato, l'ordine rimane invariato:

```
>>> d[22] = 'twenty-two'
>>> d
OrderedDict([(11, 'undici'), (22, 'twenty-two'), (33, 'trentatre')])
```

Se, invece, viene cancellato e poi inserito, viene posizionato in coda:

```
>>> del d[22]
>>> d[22] = 'twenty-two'
>>> d
OrderedDict([(11, 'undici'), (33, 'trentatre'), (22, 'twenty-two')])
```

I dizionari ordinati hanno tutti i metodi dei dizionari classici, e in più il metodo `OrderedDict.move_to_end()`. Questo consente di spostare un elemento in ultima o in prima posizione, a seconda che il suo argomento opzionale `last` valga `True` oppure `False`. Per default si ha `last=True`:

```
>>> d.move_to_end(11)
>>> d
OrderedDict([(33, 'trentatre'), (22, 'twenty-two'), (11, 'undici')])
>>> d.move_to_end(11, False)
>>> d
OrderedDict([(11, 'undici'), (33, 'trentatre'), (22, 'twenty-two')])
```

Mentre nei dizionari classici il metodo `dict.popitem()` restituisce e rimuove un coppia chiave-valore in modo non deterministico, nei dizionari ordinati `OrderedDict.popitem()` restituisce e rimuove l'ultimo valore inserito oppure il primo, a seconda che il suo parametro `last` valga `True` oppure `False`. Per default si ha `last=True`:

```
>>> d.popitem()
(22, 'twenty-two')
>>> d
OrderedDict([(11, 'undici'), (33, 'trentatre')])
>>> d.popitem(False)
(11, 'undici')
>>> d
OrderedDict([(33, 'trentatre')])
```

Persistenza e serializzazione degli oggetti

Se un oggetto viene salvato in un dispositivo non volatile, in modo che sia disponibile anche dopo che il programma ha terminato la sua esecuzione, si dice che è *persistente nel tempo*. Capita spesso di avere la necessità di rendere degli oggetti persistenti, perché, ad esempio, si vuole riutilizzarli nel corso di una successiva esecuzione del programma, oppure perché devono essere utilizzati da altri programmi in altri momenti.

La persistenza può essere realizzata in vari modi, ad esempio salvando l'oggetto in un *database*, oppure trasformandolo in una sequenza di byte che viene poi salvata su file. In quest'ultimo caso il processo di trasformazione dell'oggetto in stream di byte e successivo salvataggio è chiamato *serializzazione*, e si dice che l'oggetto viene *serializzato* in un *file di oggetti*. Il processo inverso, che consente di ricostruire un oggetto a partire dal file di oggetti, è detto *de-serializzazione*.

NOTA

La serializzazione è usata frequentemente in programmazione, e probabilmente senza saperlo abbiamo dei programmi che ne fanno uso.

Pensiamo, ad esempio, a un gioco: tipicamente, nel momento in cui viene terminata l'esecuzione, viene offerta all'utente la possibilità di salvare lo stato di gioco attuale, per poi ricaricarlo in un secondo momento, quando l'utente riprenderà a giocare. Questo processo, di solito, viene implementato usando il meccanismo della serializzazione.

In Python la serializzazione è chiamata *pickling*, nome che deriva dal verbo inglese *to pickle*, utilizzato per indicare il processo di conservazione sotto aceto. Il modulo della libreria standard che consente di effettuare la serializzazione si chiama `pickle`. Se vogliamo approfondire l'argomento possiamo consultare la documentazione ufficiale, alla pagina <http://docs.python.org/3/library/pickle.html>.

Ciò di cui vogliamo parlare in questa sezione è un modulo basato su `pickle`, che consente di effettuare la serializzazione utilizzando una interfaccia analoga a quella dei dizionari. Questo modulo si chiama `shelve`, e i file di oggetti da esso creati sono sostanzialmente dei dizionari persistenti, che vengono chiamati *shelf*. Questi vengono creati e aperti utilizzando la funzione `shelve.open()`:

```
>>> import shelve
>>> shelf = shelve.open('mysshelf')
>>> shelf['name'] = 'Albert Einstein'
>>> shelf['city'] = 'Ulm '
>>> shelf.close()
```

Come possiamo vedere, sono stati creati tre file fisici:

```
$ ls
mysshelf.bak mysshelf.dat mysshelf.dir
```

Per poter effettuare la deserializzazione dobbiamo aprire il file di oggetti senza utilizzare il suffisso:

```
$ python -c "import shelve; shelf = shelve.open('mysshelf'); print(shelf['name'])"
Albert Einstein
```

Questa è stata solo una breve introduzione al modulo `shelve`. Se vogliamo approfondire l'argomento, possiamo consultare la documentazione ufficiale: <http://docs.python.org/3/library/shelve.html>.

Le sequenze

Come abbiamo già detto nel precedente capitolo, gli oggetti che fanno parte della categoria delle *sequenze* sono le *stringhe*, le *tuple* e le *liste*. Le sequenze sono degli oggetti iterabili che rappresentano dei contenitori di lunghezza arbitraria. La funzione built-in `len()` restituisce la lunghezza della sequenza che riceve come argomento:

```
>>> len('python')
6
```

Operazioni comuni alle sequenze

In questa sezione riassumeremo le operazioni comuni alle sequenze.

Indicizzazione e slicing

Gli elementi delle sequenze 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. L'operazione che consente di ottenere un elemento della sequenza, utilizzando come chiave di ricerca l'indice, è detta *indicizzazione*:

```
>>> 'python'[1]
'y'
>>> ['p', 'y', 't', 'h', 'o', 'n'][1]
'y'
>>> ('p', 'y', 't', 'h', 'o', 'n')[1]
'y'
```

NOTA

Solo le liste supportano l'indicizzazione alla sinistra di un assegnamento:

```
>>> mylist = [1, 2, 3]
>>> mylist[2] = 33 # Le liste sono oggetti mutabili
>>> mylist
[1, 2, 33]
>>> s = 'python'
>>> s[0] = 'P' # Le stringhe e le tuple sono oggetti immutabili
Traceback (most recent call last):
...
TypeError: 'str' object does not support item assignment
```

Data una sequenza `seq`, lo *slicing* `seq[i:j]` restituisce una nuova sequenza contenente gli elementi di `seq` compresi tra gli indici `i` e `j` (escluso):

```
>>> 'python'[1:3]
'yt'
>>> ['p', 'y', 't', 'h', 'o', 'n'][1:3]
['y', 't']
>>> ('p', 'y', 't', 'h', 'o', 'n')[1:3]
('y', 't')
```

Nello slicing è possibile utilizzare degli indici negativi. L'indice `-1` identifica l'ultimo elemento della sequenza, l'indice `-2` il penultimo, e via dicendo:

```
>>> s[-1], s[-2] # L'indice '-1' identifica l'ultimo elemento, l'indice '-2' il penultimo
('n', 'o')
```

Quando viene indicato solamente un indice, la sequenza viene attraversata interamente a partire da esso:

```
>>> s[1:] # Dal primo elemento sino all'ultimo (incluso)
'ython'
>>> s[:-1] # Dal primo elemento sino all'ultimo (escluso)
'pytho'
```

Se non viene specificato nemmeno un indice, viene restituita una copia della sequenza:

```
>>> s[:]
'python'
```

Con lo slicing è possibile utilizzare un terzo indice, che per default vale `1`, in modo da stabilire il *passo*, ovvero il salto da compiere tra due elementi consecutivi:

```
>>> s = 'python programming language'
>>> s[::1] # Questo è il comportamento di default, equivalente a `s[:]`
'python programming language'
>>> s[::2] # Estrae dalla sequenza un elemento ogni due
'pto rgamn agae'
>>> s[::3] # Estrae dalla sequenza un elemento ogni tre
'ph oai na'
>>> s[2:10:3] # Estrae dallo slice `seq[2:10]` un elemento ogni tre
'tnr'
```

Quando il terzo indice è negativo, la sequenza viene attraversata al contrario, per cui il primo indice deve essere maggiore o uguale al secondo:

```
>>> s[5:1:-2] # Un elemento ogni due, dall'indice 5 a 1 (escluso)
'nh'
>>> s[6:-1] # Dall'ultimo elemento sino a quello di indice 6 (escluso)
'egaugnal gnimmargorp'
```

Per ottenere una copia invertita della sequenza, si può eseguire uno slicing nel seguente modo:

```
>>> s[::-1] # In questo caso, quindi, viene restituita una copia di `s` con elementi invertiti
'egaugnal gnimmargorp nohtyp'
```

Per iterare su una sequenza al contrario si può utilizzare sia lo slicing appena visto che la funzione built-in `reversed()`, la quale restituisce un iteratore che itera sulla sequenza al contrario:

```
>>> for item in ['a', 'b', 'c'][::-1]:
...     print(item)
...
c
b
a
>>> for item in reversed(['a', 'b', 'c']):
...     print(item)
...
c
b
a
```

Lo slicing non solleva eccezioni se si utilizzano indici fuori dall'intervallo:

```
>>> s[-1000:1000] # Indici fuori dall'intervallo
'python programming language'
>>> s[6:1] # Il primo indice è maggiore del secondo, con passo pari a 1
''
>>> s[1:5:-1] # Il primo indice è minore del secondo, con passo pari a -1
''
```

Lo slicing può essere eseguito anche tramite la classe built-in `slice`, la quale viene chiamata con l'argomento obbligatorio `stop`, e ha due argomenti opzionali `start` e `step`:

```
>>> s = slice(4)
>>> type(s)
<class 'slice'>
>>> s.start, s.stop, s.step
(None, 4, None)
```

```
>>> mylist = list(range(10))
>>> mylist[s] # Equivale a `mylist[:4]`
[0, 1, 2, 3]
>>> s = slice(3, 10, 2)
>>> s.start, s.stop, s.step
(3, 10, 2)
>>> mylist[s] # Equivale a `mylist[3:4:2]`
[3, 5, 7, 9]
```

Dato un oggetto `s` di tipo `slice`, il metodo `s.indices(length)` calcola gli indici che avrebbe `s` se fosse applicata a una sequenza di lunghezza `length`. Questo metodo restituisce una tupla di tre interi, che rappresentano, rispettivamente, gli indici `start`, `stop` e `step`:

```
>>> s = slice(None, None, -1)
>>> s = slice(7)
>>> s.indices(5)
(0, 5, 1)
>>> s.indices(9)
(0, 7, 1)
>>> s = slice(None, None, -1)
>>> s.indices(3)
(2, -1, -1)
>>> s.indices(7)
(6, -1, -1)
```

Lo slicing delle liste può comparire alla sinistra di un assegnamento:

```
>>> mylist = ['a', 'b', 100, (1, 2, 3)]
>>> mylist[0:2] = ['python'] # Inserisce 'python' al posto di mylist[0:2]
>>> mylist
['python', 100, (1, 2, 3)]
>>> mylist[0:1] = 'python' # Spacchetta la stringa 'python'
>>> mylist
['p', 'y', 't', 'h', 'o', 'n', 100, (1, 2, 3)]
```

e può essere processato dall'istruzione `del`:

```
>>> del mylist[:6]
>>> mylist
[100, (1, 2, 3)]
```

Concatenazione e ripetizione

La *concatenazione* unisce due o più sequenze creando una nuova sequenza:

```
>>> 'python' + '3'
'python3'
>>> ['p', 'y', 't', 'h', 'o', 'n'] + [3]
['p', 'y', 't', 'h', 'o', 'n', 3]
```

```
>>> ('p', 'y', 't', 'h', 'o', 'n') + (3,)
('p', 'y', 't', 'h', 'o', 'n', 3)
```

La *ripetizione* riproduce una sequenza per un certo numero di volte:

```
>>> 'python' * 3
'pythonpythonpython'
>>> ['p', 'y', 't', 'h', 'o', 'n'] * 3
['p', 'y', 't', 'h', 'o', 'n', 'p', 'y', 't', 'h', 'o', 'n', 'p', 'y', 't', 'h', 'o', 'n']
>>> ('p', 'y', 't', 'h', 'o', 'n') * 3
('p', 'y', 't', 'h', 'o', 'n', 'p', 'y', 't', 'h', 'o', 'n', 'p', 'y', 't', 'h', 'o', 'n')
```

Oltre alle operazioni di concatenazione e ripetizione semplice, le sequenze supportano gli *augmented assignment* `+=` e `*=`.

Come abbiamo visto nella sezione *Modifica di oggetti mutabili* del precedente capitolo, il comportamento di questi ultimi è diverso a seconda che l'oggetto a cui l'etichetta fa riferimento sia mutabile o immutabile. Infatti, mentre la concatenazione e la ripetizione creano sempre dei nuovi oggetti, gli *augmented assignment* modificano l'oggetto se questo è mutabile:

```
>>> mylist = [1, 2, 3] # Le liste sono oggetti mutabili
>>> id(mylist)
3071513292
>>> mylist *= 3 # Non è la stessa cosa che fare `mylist = mylist * 3`
>>> mylist, id(mylist) # Infatti è stata modificata la lista
([1, 2, 3, 1, 2, 3, 1, 2, 3], 3071513292)
>>> t = (1, 2, 3) # Le tuple sono oggetti immutabili
>>> id(t)
3072199452
>>> t += (4,) # Equivale a `t = t + (4,)`
>>> t, id(t) # Infatti è stato creato un nuovo oggetto
((1, 2, 3, 4), 3072198332)
```

Test di verità

Come abbiamo già detto, una sequenza vuota è considerata `False`, altrimenti è considerata `True`:

```
>>> bool(""), bool(' ') # Stringa vuota e una stringa contenente uno spazio
(False, True)
>>> bool([]), bool(['']) # Lista vuota e lista contenente una stringa vuota
(False, True)
>>> bool(()), bool(("",)) # Tupla vuota e una tupla contenente una stringa vuota
(False, True)
```

Stringhe

Le stringhe sono presenti con varie forme in tutti i linguaggi di

programmazione. È difficile pensare a un programma che non ne faccia uso o a un programmatore che non le abbia mai usate e non sappia cosa siano. Nonostante ciò, e nonostante appaiano a prima vista come degli oggetti molto semplici, si commettono tantissimi errori nell'utilizzarle.

Questi errori sono originati da una scarsa conoscenza dei meccanismi di codifica. Se ci pensiamo, infatti, il legame che intercorre tra i caratteri e la loro rappresentazione binaria è inevitabilmente molto stretto, visto che in tutte le operazioni di input/output i dati sono rappresentati ad alto livello con delle stringhe, come abbiamo visto nel caso dei file, dove il metodo `write()` prende una stringa come argomento e il metodo `read()` restituisce una stringa. Il primo passo consiste, quindi, nel capire come funzionano i meccanismi di codifica dei caratteri, e come vengono gestiti da Python.

I sistemi di codifica dei caratteri

Un *sistema di codifica dei caratteri* (in inglese *character encoding system*) è uno schema che associa a ogni carattere di un certo insieme un codice numerico univoco. Come sappiamo, infatti, nel mondo digitale i dati sono rappresentati in formato binario, per cui, se vogliamo memorizzare o trasmettere un carattere tramite un sistema digitale, dobbiamo necessariamente rappresentarlo con una sequenza di bit. Ad esempio, se volessimo trasmettere dei caratteri e potessimo inviare solo due bit per volta, potremmo decidere di utilizzare le sequenze 00, 01, 10 e 11 per rappresentare, rispettivamente, i caratteri A, B, C e D. Se il ricevente conosce la nostra codifica, allora può compiere l'operazione inversa, chiamata *decodifica* (in inglese *decoding*): quando riceverà la sequenza di bit 00 saprà che il mittente gli ha inviato il carattere A, quando riceverà 01 saprà che gli è stato inviato il carattere B, e via dicendo. Ma, se il ricevente non conosce la codifica utilizzata per trasmettere i dati, riceverà delle sequenze di bit alle quali non saprà attribuire alcun significato. Quest'ultima considerazione non è affatto banale, ma rappresenta un problema frequente. Si pensi, infatti, che il *testo semplice* (in inglese *plain text*) non contiene alcun metadato, e quindi non fornisce a chi deve leggerlo alcuna informazione sulla codifica. Questo è il motivo per cui spesso nelle pagine web, nelle e-mail o nei file di testo si vedono dei caratteri strani laddove ci sarebbero dovute essere delle lettere accentate.

NOTA

Visto che i file di testo non contengono alcun metadato, tipicamente le

applicazioni che utilizziamo per leggerli non possono fare altro che provare a indovinare la codifica. I caratteri non accentati e i simboli di punteggiatura sono codificati allo stesso modo in quasi tutti i sistemi di codifica, ma non possiamo dire la stessa cosa per le lettere accentate, per cui, se il vostro *text reader* non azzecca la codifica, solitamente si ottiene un risultato di questo tipo:

```
$ cat myfile # Scritto con codifica iso8859_1 ma letto con UTF-8
Quanto ? bello Python! :)
```

Spesso per risolvere questo problema si dichiara la codifica all'interno del testo. Nel caso delle pagine web, ad esempio, si inserisce nel testo un apposito tag html chiamato *meta*, come il seguente:

```
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
```

Grazie a questa convenzione il browser per scoprire la codifica può leggere il tag *meta*, e, così facendo, è in grado di visualizzare i caratteri in modo corretto. A questo punto è lecito chiedersi come sia possibile vedere nelle pagine web dei caratteri strani, visto che il browser conosce la codifica utilizzata. È possibile, ad esempio, che talvolta il *webmaster*, non sapendo ciò che fa, indichi una codifica errata. Ecco uno dei tanti motivi per cui vale la pena studiare i meccanismi di codifica.

Come abbiamo visto nella sezione *Esecuzione del codice Python da file* del [Capitolo 1](#), anche Python adotta la stessa strategia che si utilizza per le pagine html. Per default, infatti, Python 3 legge il codice usando la codifica UTF-8, per cui, se editiamo i nostri moduli usando una codifica differente, dobbiamo informarlo inserendo un commento speciale, contenente `coding:nome` o `coding=nome`.

NOTA

Sia il browser che l'interprete Python, per poter decodificare il testo in modo corretto, devono leggere una parte del contenuto (il tag *meta* o il commento speciale). Ma come possono farlo, se ancora non conoscono il sistema di codifica? Ebbene sì, questo è un circolo vizioso, ma, se leggeremo le prossime pagine, avremo il piacere di trovare da soli una risposta.

La rappresentazione dei caratteri in formato digitale

Nei sistemi digitali l'unità di misura utilizzata per quantificare i dati è chiamata *byte*. Questo termine è stato utilizzato per la prima volta nel 1956 da Werner Buchholz per indicare *a bite of bit*, ovvero *un boccone di bit*. Si decise, però, di utilizzare la parola *byte* per evitare confusione, visto che *bite* è troppo simile a *bit*.

Per capire quanto la rappresentazione digitale dei caratteri sia legata al concetto di *byte*, riportiamo quanto scritto da Werner Buchholz et al. nel 1962, in *Planning a Computer System – Project Stretch*:

Il termine byte indica un gruppo di bit usati per codificare un carattere, o il numero di bit trasmessi in parallelo verso e da delle unità di input-output. Usiamo il termine byte, piuttosto che carattere, perché un dato carattere può essere rappresentato in differenti applicazioni da più di un codice, e differenti codici possono usare differenti numeri di bit (cioè differenti dimensioni del byte). Nelle trasmissioni in input-output il raggruppamento dei bit può essere completamente arbitrario e non avere alcuna relazione con i caratteri. (Il termine deriva da bite, ma quest'ultimo è stato riscritto in modo da evitare che avvengano accidentali trasformazioni che portano a scrivere bit.)

In quegli anni le architetture dei computer erano tipicamente a 6 bit, per cui si potevano rappresentare al massimo 64 caratteri. Venivano codificati solo i caratteri rappresentativi delle cifre decimali, dei simboli di punteggiatura e delle lettere dell'alfabeto inglese, niente di più. Ad esempio, i codici 000000 e 000011 venivano usati per rappresentare, rispettivamente, il carattere 0 e il carattere 3, il codice 011000 per rappresentare il carattere H e il codice 111111 per il carattere di punto esclamativo. Quindi la stringa 30 veniva codificata con una sequenza di 2 byte, il byte 000011 rappresentativo del carattere 3 e quello 000000 rappresentativo del carattere 0, mentre la stringa HI! veniva codificata con la sequenza di byte 011000, 011001 e 111111.

Con soli 64 codici a disposizione non si riusciva a rappresentare neppure la versione minuscola delle lettere dell'alfabeto inglese, per cui, nel 1960, tenuto conto che nell'immediato futuro la maggior parte delle architetture di computer sarebbe stata a 8 bit, l'American Standards Association iniziò a lavorare alla definizione di un nuovo sistema di codifica a 8 bit. Questo portò, nel 1963, alla nascita della prima versione del *sistema di codifica ASCII*, e da quel momento si iniziò a utilizzare il termine *byte* per indicare proprio una sequenza di 8 bit.

Studiare lo standard ASCII dopo mezzo secolo dalla sua definizione non è affatto anacronistico come si potrebbe ingenuamente pensare. Si consideri che, sino al dicembre 2007, questo sistema di codifica era il più usato nel web, e il suo posto è stato preso dal sistema di codifica UTF-8, che a sua volta è basato sul sistema ASCII. Già dalle prossime pagine ci accorgeremo che non è possibile lavorare con stringhe e byte se non si conosce la codifica ASCII. Questo standard sarà il nostro pane quotidiano.

American Standard Code for Information Interchange

Lo standard ASCII definisce un sistema di codifica che consente di rappresentare 128 caratteri, non 256 come ci si sarebbe aspettato da un sistema di codifica a 8 bit. Si decise, infatti, di utilizzare solamente 7 bit per la codifica, riservando l'utilizzo dell'ottavo per effettuare un controllo degli errori tramite test di parità. I codici binari partono quindi da 00000000 (il numero 0 in formato decimale) sino ad arrivare a 01111111 (il numero 127 in formato decimale).

NOTA

Il sistema ASCII è talmente importante che i caratteri da esso rappresentati vengono comunemente chiamati *caratteri ASCII*. Ad esempio, i caratteri `c`, `i`, `a` e `o` fanno parte del sistema ASCII, per cui si dice che la stringa `'ciao'` è composta da caratteri ASCII.

Per semplicità, d'ora in avanti mostreremo i codici solamente in formato decimale. Ad esempio, non indicheremo il codice del carattere `A` con 01000001, ma piuttosto con il corrispondente numero ordinale 65.

La funzione built-in `ord()` prende come argomento una stringa composta da un solo carattere e restituisce il codice ordinale di quel carattere. Il carattere `B`, ad esempio, è codificato con il codice 66:

```
>>> ord('B')
66
```

La funzione built-in `chr()` esegue l'operazione inversa, ovvero prende come argomento un numero e restituisce la stringa contenente il carattere a esso associato:

```
>>> chr(66)
'B'
```

A questo punto, prima di andare avanti, è necessario fare una precisazione. Il termine *carattere* spesso viene usato in modo scorretto, pensando che a esso corrisponda sempre un simbolo grafico. Questo errore è dovuto al fatto che siamo abituati ad associare i caratteri alle lettere del nostro alfabeto, ai numeri e ai simboli utilizzati per la punteggiatura. Per ognuno di essi, infatti, esiste un simbolo grafico corrispondente. La parola *carattere*, in realtà, ha un'accezione più ampia. Non tutti i caratteri, infatti, nascono per essere stampati. Se, ad esempio, provassimo a stampare i caratteri corrispondenti ai codici 8 e 9, non vedremmo nulla:

```
>>> for i in range(8, 10):
...     print(chr(i))
...
>>>
```

Questi non sono caratteri magici, ma dei comunissimi caratteri che usiamo ogni giorno. Il carattere con codice 9, ad esempio, è una tabulazione, e possiamo inserirlo nel testo premendo il tasto **TAB**. La sua funzione è quella di aggiungere un certo numero di spazi vuoti per indentare il testo, senza doverlo fare con la barra spaziatrice. I caratteri di questo tipo, ai quali non è associato alcun simbolo grafico, vengono detti *caratteri di controllo*, in modo da distinguerli dagli altri, detti *caratteri stampabili*.

I caratteri di controllo hanno codici che vanno da 0 a 31, più il codice 127, mentre i caratteri stampabili hanno codici che vanno da 32 a 126, e sono i seguenti:

```
>>> for i in range(32, 127):
...     print(chr(i), end=' ')
...
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ]
^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

Come possiamo vedere, non vi è neppure un lettera accentata, per cui con la codifica ASCII non è possibile scrivere in italiano, francese, spagnolo, tedesco, per non parlare delle lingue orientali.

In realtà si scrive ugualmente anche con il solo insieme di caratteri ASCII. Ad esempio, in italiano ci si arrangia scrivendo le lettere accentate come la combinazione della corrispondente lettera non accentata seguita dall'apostrofo (il carattere `è` viene scritto come `e'`). In francese si utilizza il carattere `ç` al posto della *cédille* (`ç`), e via dicendo per le altre lingue.

È facile immaginare che non tardarono ad arrivare nuovi standard e sistemi di codifica, ma quasi sempre finalizzati a sopperire alla necessità di una particolare lingua o insieme di lingue. Un esempio è lo standard ISO 8859, che definisce 15 sistemi di codifica (ISO 8859-*n*) a 8 bit, ciascuno dei quali consente di rappresentare 256 caratteri (vengono utilizzati tutti e otto i bit, diversamente dal sistema ASCII). Di questi 256, i primi 128 sono i caratteri ASCII, mentre i restanti 128 sono specifici di ciascuno dei 15 diversi sistemi. Ad esempio, nel sistema di codifica ISO 8859-1, chiamato Latin-1, i 128 caratteri aggiuntivi consentono di soddisfare le esigenze della maggior parte delle lingue europee occidentali, poiché comprendono tutti i caratteri accentati della lingua italiana, della lingua tedesca, una parte di quelli della lingua francese, e via dicendo.

La svolta arrivò alla fine del 1987, quando si iniziò a parlare di uno standard di codifica universale, chiamato Unicode. Prima di parlare dello standard Unicode è però necessario introdurre un tipo di dato built-in del quale non abbiamo ancora parlato, strettamente legato ai caratteri ASCII. Si tratta delle stringhe di byte.

Stringhe di byte

In Python le stringhe non sono rappresentate dal solo tipo `str`, ma anche dal tipo `bytes`. Le stringhe di tipo `str` sono dette *stringhe di testo*, mentre quelle di tipo `bytes` sono dette *stringhe di byte*.

I letterali delle stringhe di byte si distinguono da quelli delle stringhe di testo per via della presenza della lettera `b`, scritta indifferentemente in minuscolo o maiuscolo:

```
>>> st = " # Stringa di testo vuota
>>> sb = b" # Stringa di byte vuota
>>> type(st), type(sb)
(<class 'str'>, <class 'bytes'>)
```

Gli elementi delle stringhe di byte, come si evince dal termine, sono dei byte. Per rappresentare un byte si utilizza la notazione `\xhh`, dove `hh` è il valore del

byte espresso con due cifre esadecimali. Ad esempio, il byte 00001100 ha valore esadecimale 0xc:

```
>>> hex(0b00001100)
'0xc'
```

per cui viene rappresentato con b'\x0c':

```
>>> s = b'\x0c' # Stringa composta dal byte 00001100 (0xc in esadecimale)
>>> s
b'\x0c'
>>> len(s) # La stringa b'\x0c' contiene un solo byte, quindi ha lunghezza 1
1
```

NOTA

Tra breve vedremo che la sintassi \xhh fa parte di un insieme di rappresentazioni simboliche dei caratteri e dei byte, chiamate *sequenze di escape*.

Nelle stringhe di testo, quando si accede singolarmente agli elementi tramite indicizzazione, iterazione o spaccettamento, si ottiene una stringa:

```
>>> s = 'ab'
>>> s[0]
'a'
>>> for item in s:
...     print(item)
...
a
b
>>> x, y = s
>>> x
'a'
```

Nelle stringhe di byte si ottiene, invece, un intero, corrispondente al valore del byte:

```
>>> sb = b'\x0c\x00\x1f' # Stringa di byte di tre elementi
>>> sb[0] # Restituisce un intero pari al valore del byte (0xc -> 12)
12
>>> for item in sb:
...     print(item)
...
12
```

```
0
31
>>> x, y, z = sb
>>> x
12
```

Se si utilizza lo slicing, allora le stringhe di byte si comportano in modo identico alle stringhe di testo, poiché danno come risultato una stringa:

```
>>> sb[0:0]
b''
>>> sb[0:1]
b'\x0c'
>>> sb[0:2]
b'\x0c\x00'
```

Se il valore di un byte corrisponde al codice di un carattere ASCII (stampabile o rappresentabile), allora è possibile far riferimento al byte utilizzando il carattere piuttosto che la notazione `\xhh`:

```
>>> s = b'abc' # Creo una stringa di byte utilizzando i caratteri ASCII
>>> for item in s:
...     print(item, hex(item))
...
97 0x61
98 0x62
99 0x63
```

In questi casi, inoltre, la shell interattiva rappresenta il byte sempre con il corrispondente carattere ASCII piuttosto che con la notazione `\xhh`:

```
>>> b'\x61\x62\x63'
b'abc'
```

Infine, a differenza delle stringhe di testo, quando le stringhe di byte vengono stampate viene mostrato il letterale e non il contenuto della stringa:

```
>>> print(b'ciao')
b'ciao'
>>> print('ciao')
ciao
```

Lo standard Unicode

Lo standard Unicode consente di codificare più di un milione di caratteri, assegnando a ciascuno di essi un codice unico. Al giorno d'oggi sono stati

assegnati circa 110.000 codici ad altrettanti caratteri.

Nella terminologia Unicode i codici sono chiamati *code point* e vengono indicati con `U+hex_code`, dove `hex_code` è il codice in formato esadecimale, scritto con almeno quattro cifre. Ad esempio, alla lettera greca α è assegnato il codice decimale 945, ovvero `0x3b1` esadecimale, quindi il relativo code point è `U+03B1`.

La mappatura dei caratteri Unicode ha come base quella dei caratteri ASCII, nel senso che i codici che vanno da 0 a 127 sono assegnati ai rispettivi caratteri ASCII, per cui, ad esempio, il codice 65, al quale corrisponde il code point `U+0041`, è assegnato alla lettera A.

Lo standard Unicode assegna ai caratteri delle *proprietà* (*character properties*). Esempi di proprietà sono il nome del carattere e la categoria generale di appartenenza. Ad esempio, il carattere A ha come nome `LATIN CAPITAL LETTER A` e come categoria `Lu` (Letter, uppercase).

Le proprietà dei caratteri Unicode sono elencate in un database di nome UCD (Unicode Character Database), che possiamo consultare alla pagina web <http://www.unicode.org/ucd/>. La libreria standard fornisce l'accesso all'UCD tramite il modulo `unicodedata`:

```
>>> import unicodedata
>>> unicodedata.name('A')
'LATIN CAPITAL LETTER A'
>>> unicodedata.category('A')
'Lu'
```

Per maggiori informazioni sul modulo `unicodedata` possiamo consultare la documentazione ufficiale, alla pagina web <http://docs.python.org/3/library/unicodedata.html>.

Sistemi di codifica che si basano sullo standard Unicode

Lo standard Unicode si limita a definire una mappatura tra codici e caratteri, ma non stabilisce come effettuare la codifica binaria. Questo significa che i sistemi di codifica basati sullo standard Unicode sono liberi di decidere con quanti byte codificare ogni carattere. Ad esempio, nel sistema UTF-32 ogni carattere è codificato con 4 byte, nel sistema UTF-16 una parte dei caratteri (quelli con codici che vanno da 0 a 65535) è codificata con 2 byte e la restante parte con 4 byte, mentre nel sistema UTF-8 i caratteri ASCII sono codificati con 1 byte e i restanti con 2 o più byte.

La sigla UTF è acronimo di *Unicode Transformation Format*. I numeri 32, 16 e 8 indicano la larghezza base della codifica in termini di bit: 32 (4 byte) nel sistema UTF-32, 16 o multipli di 16 (2 o 4 byte) nel sistema UTF-16, e infine 8 o multipli di 8 (uno o più byte) nel sistema UTF-8. Come detto più volte, Python 3 utilizza UTF-8 come sistema di codifica di default.

Vediamo un esempio di codifica dello stesso carattere nei tre sistemi. Consideriamo, a tale scopo, il carattere A, che come sappiamo ha codice decimale 65, e quindi rappresentazione binaria 1000001. Nel sistema UTF-32 viene codificato con 4 byte: uno è 01000001 (`\x41`) e i restanti tre tutti 00000000 (`\x00`). Nel sistema UTF-16 è codificato con 2 byte: uno è 01000001 (`\x41`) e l'altro 00000000 (`\x00`). Nel sistema UTF-8 è invece codificato con il solo byte 01000001 (`\x41`). Abbiamo detto che lo standard Unicode adotta la stessa mappatura dei caratteri ASCII utilizzata dal sistema di codifica ASCII, e che il sistema UTF-8 codifica i caratteri ASCII con un solo byte. Questo significa che il sistema UTF-8 codifica i 128 caratteri ASCII in modo identico al sistema di codifica ASCII, ovvero stessa mappatura tra codice e carattere e stessa rappresentazione binaria. Infatti forse avevamo già notato che il carattere A in UTF-8 è codificato proprio come nel sistema di codifica ASCII, cioè con il solo byte `\x41` (01000001).

NOTA

È conveniente codificare il testo contenente prevalentemente caratteri ASCII in UTF-8 piuttosto che in UTF-16 o UTF-32, poiché è evidente che in UTF-8 si ha in media una dimensione in byte inferiore a quella che si avrebbe se lo stesso testo venisse codificato in UTF-16 o UTF-32. Uno studio condotto da W3Techs (<http://w3techs.com/>) nel dicembre 2011 ha mostrato che i contenuti, in oltre il 56% di tutti i siti web, sono scritti in lingua inglese, e questo significa che nel web il testo contiene prevalentemente caratteri ASCII. Questo è uno dei motivi per cui il sistema UTF-8 è il più diffuso nel web.

Le stringhe di testo in Python 3 sono sequenze di caratteri Unicode. Data una stringa di testo, il metodo `str.encode()` ci consente di ottenere la corrispondente stringa di byte. Se a `str.encode()` non passiamo il nome di una codifica, viene restituita la corrispondente stringa di byte codificata in UTF-8:

```
>>> 'A'.encode()
b'A'
>>> 'è'.encode()
b'\xc3\xa8'
>>> 'cioè'.encode()
b'cio\xc3\xa8'
```

Abbiamo detto che nei sistemi di codifica UTF-32 e UTF-16 il carattere A è codificato, rispettivamente, con 4 e 2 byte:

```
>>> 'A'.encode('UTF-32')
b'\xff\xfe\x00\x00A\x00\x00\x00'
>>> 'A'.encode('UTF-16')
b'\xff\xfeA\x00'
```

In effetti è codificato con `b'A\x00\x00\x00'` in UTF-32 e con `b'A\x00'` in UTF-16. Come possiamo osservare, però, vi sono dei byte aggiuntivi in testa alla stringa: i 4 byte `\xff\xfe\x00\x00` in UTF-32 e i 2 byte `\xff\xfe` in UTF-16. Questi byte sono una caratteristica delle codifiche UTF-32 e UTF-16, e rappresentano un carattere speciale chiamato *Byte Order Mark* (BOM).

Il BOM serve per indicare l'ordine in cui sono disposti in sequenza i byte, poiché questa informazione è indispensabile per il calcolo del valore del codice. Se i byte sono ordinati in modo che la sequenza dei bit corrisponda alla rappresentazione binaria del valore rappresentato, si dice che l'ordine è *big endian*. Ad esempio, se consideriamo la codifica del carattere A (codice 65) in UTF-16, l'ordine big endian dei byte è `\x00\x41`, perché `00000000 01000001` in notazione decimale è proprio 65:

```
>>> 0b0000000001000001
65
```

Se, invece, i byte sono invertiti, allora la sequenza di bit non fornisce la rappresentazione binaria del valore rappresentato, e in questo caso si dice che l'ordine è *little endian*. Se consideriamo, ancora una volta, la codifica del carattere A in UTF-16, l'ordine little endian è `\x41\x00`, corrispondente in binario a `01000001 00000000`, quindi al valore decimale 16640 piuttosto che a 65:

```
>>> 0b0100000100000000
16640
```

Per indicare che l'ordine è little endian, si utilizza il BOM `\xff\xfe\x00\x00` in UTF-32 e il BOM `\xff\xfe` in UTF-16, mentre per indicare che è big endian si utilizza il BOM `\xfe\xff\x00\x00` in UTF-32 e il BOM `\xfe\xff` in UTF-16.

NOTA

Entrambi i sistemi dispongono delle varianti *-LE* (little endian) e *-BE* (big endian), nelle quali non c'è bisogno di utilizzare il BOM visto che l'ordine dei byte è indicato nel nome della codifica:

```
>>> 'A'.encode('UTF-16-LE'), 'A'.encode('UTF-16-BE')
(b'A\x00', b'\x00A')
```

Il BOM viene specificato solamente una volta, all'inizio della stringa:

```
>>> 'python'.encode('UTF-16')
b'\xff\xfe\x00p\x00y\x00t\x00h\x00o\x00n\x00'
```

NOTA

In Python 2, a differenza di Python 3, il tipo `str` non rappresenta stringhe di caratteri Unicode ma stringhe di byte. Se il valore del byte coincide con il codice di un carattere ASCII, allora viene visualizzato con il carattere corrispondente:

```
>>> s = 'abè' # Python 2
>>> s
'ab\xc3\xa8'
>>> len(s) # In Python 2 len() restituisce il numero di byte
4
```

Il tipo `str` di Python 2 è però ancora diverso dal tipo `bytes` di Python 3:

```
>>> s[2] # In Python 2 l'indicizzazione non restituisce un intero
'\xc3'
```

Se in Python 2 vogliamo creare una stringa di caratteri Unicode dobbiamo utilizzare il tipo `unicode` e non il tipo `str`. I letterali delle stringhe di tipo `unicode` si creano antepoendo una `u` ai letterali classici:

```
>>> len(u'è'), type(u'è') # Python 2
(1, <type 'unicode'>)
```

In Python 3 la classe built-in `unicode` non esiste più, poiché le stringhe di testo sono sempre Unicode codificate per default in UTF-8 e rappresentate dal tipo

`str`. Per questo motivo si era deciso di abbandonare la sintassi esplicita dei letterali Unicode, e infatti nelle versioni di Python 3 inferiori alla 3.3 questa sintassi non è consentita:

```
>>> u'python' # Python 3.2
File "<stdin>", line 1
u'python'
^
SyntaxError: invalid syntax
```

Per ridurre il numero di incompatibilità tra Python 2 e Python 3 si è però deciso di reintrodurre questa sintassi con Python 3.3 (vedi PEP-0414). Il risultato, ovviamente, è una stringa di tipo `str`:

```
>>> s = u'python' # Python 3.3 o superiore
>>> s, type(s)
('python', <class 'str'>)
```

Conversioni tra stringhe di testo e stringhe di byte

Le stringhe di testo e di byte non vengono mai convertite l'una nell'altra in modo automatico:

```
>>> b'stringa di byte' + 'stringa di testo'
Traceback (most recent call last):
...
TypeError: can't concat bytes to str
```

Per poter eseguire delle operazioni tra i due tipi di stringhe è necessario effettuare preventivamente una conversione di uno dei due tipi nell'altro. La conversione da stringhe di testo a stringhe di byte può essere effettuata utilizzando il metodo `str.encode()`, il quale, come abbiamo visto in precedenza, per default restituisce una stringa di byte codificata in UTF-8:

```
>>> s = 'Ma quanto è bello Python 3!'
>>> s.encode() # Restituisce una stringa di byte codificata per default in UTF-8
b'Ma quanto \xc3\xa8 bello Python 3!'
```

Come ormai sappiamo, il metodo `str.encode()` prende un secondo argomento opzionale che serve per specificare una codifica differente da quella di default:

```
>>> s.encode('latin1')
b'Ma quanto \xe8 bello Python 3!'
```

NOTA

L'elenco di tutte le codifiche disponibili è riportato nella documentazione online del modulo `codecs`, che possiamo consultare alla pagina web <http://docs.python.org/3/library/codecs.html>, nella sezione *Standard Encodings*.

È possibile utilizzare anche un terzo argomento opzionale, chiamato `errors`, che consente di stabilire come gestire l'errore nel caso in cui non sia possibile convertire la stringa secondo la codifica indicata. I possibili valori di `errors` sono `'strict'`, `'ignore'`, `'replace'` o `'xmlcharrefreplace'`. Se `errors='strict'` (il caso di default) e non è possibile convertire la stringa secondo la codifica specificata, allora viene sollevata una eccezione di tipo `UnicodeEncodeError`:

```
>>> 'è'.encode('ascii') # Per default si ha `errors='strict'`
Traceback (most recent call last):
...
UnicodeEncodeError: 'ascii' codec can't encode character '\xe8' in position 0: ordinal not in range(128)
```

Gli altri casi sono illustrati di seguito:

```
>>> 'ciòè'.encode('ascii', 'ignore') # L'errore viene ignorato
b'cio'
>>> 'ciòè'.encode('ascii', 'replace') # Il carattere viene rimpiazzato
b'cio?'
>>> 'ciòè'.encode('ascii', 'xmlcharrefreplace') # Rimpiazzato con codice xml
b'cio&#232;'
```

La codifica da stringa di testo a stringa di byte può essere effettuata anche con la classe `bytes`. A differenza di `str.encode()`, l'argomento `encoding` non è opzionale, per cui è necessario specificare la codifica:

```
>>> bytes('ciòè', 'utf-8')
b'cio\xc3\xa8'
>>> bytes('ciòè', 'ascii', 'replace')
b'cio?'
```

NOTA

La classe `bytes` non accetta solo stringhe di testo come primo argomento, ma anche oggetti di altro tipo, come, ad esempio, delle sequenze di interi:

```
>>> mylist = [ord(c) for c in 'python']
>>> mylist
[112, 121, 116, 104, 111, 110]
>>> bytes(mylist)
b'python'
```

Per maggiori informazioni possiamo consultare `help(bytes)`.

Si può effettuare la conversione inversa, da stringhe di byte a stringhe di testo, sia con il metodo `bytes.decode()` sia con la classe `str`. Il metodo `bytes.decode()` è speculare a `str.encode()`, per cui prende due argomenti opzionali, `encoding` ed `errors`, che per default assumono i valori `'utf-8'` e `'strict'`:

```
>>> b'cio\xc3\xa8'.decode() # Per default utilizza la codifica UTF-8
'cioè'
>>> b'cio\xc3\xa8'.decode('ascii') # Per default utilizza `errors='strict'`
Traceback (most recent call last):
...
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3: ordinal not in range(128)
>>> b'cio\xc3\xa8'.decode('ascii', 'replace')
'cio??'
```

La classe `str` è speculare alla classe `bytes`, nel senso che prende gli argomenti `encoding` e `errors`, dei quali il primo non ha un valore di default, mentre il secondo per default vale `'strict'`:

```
>>> str(b'cio\xc3\xa8', 'utf-8')
'cioè'
>>> str(b'cio\xc3\xa8', 'ascii') # Per default si ha `errors='strict'`
Traceback (most recent call last):
...
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3: ordinal not in range(128)
```

NOTA

Se vogliamo effettuare la conversione da stringa di testo a stringa di byte usando la classe `bytes`, commettiamo un errore logico se non specifichiamo esplicitamente la codifica, poiché otteniamo una stringa di testo contenente il letterale della stringa di byte:

```
>>> str(b'ciao')
'b'ciao''
```

In base a quanto detto, ecco degli esempi di conversioni esplicite che

consentono di effettuare delle concatenazioni tra i due tipi di stringa:

```
>>> b'stringa di ' + 'byte'.encode()
b'stringa di byte'
>>> b'stringa di '.decode() + 'testo'
'stringa di testo'
>>> str(b'stringa di ', 'utf-8') + 'testo'
'stringa di testo'
>>> b'stringa di ' + bytes('byte', 'utf-8')
b'stringa di byte'
```

Sequenze di escape

Nella sezione *American Standard Code for Information Interchange* abbiamo detto che alcuni caratteri, chiamati *caratteri di controllo*, non sono stampabili, nel senso che non hanno associato alcun simbolo grafico. Per poterli inserire all'interno di una stringa è quindi necessario trovare una rappresentazione simbolica, con la quale poter fare riferimento a essi. In realtà servirebbero delle rappresentazioni simboliche anche per inserire e rappresentare alcuni caratteri stampabili all'interno delle stringhe, per varie ragioni. Ad esempio, non possiamo inserire direttamente un carattere di singolo apice all'interno di un letterale stringa delimitato da singoli apici:

```
>>> 'Partirò domani all'alba'
File "<stdin>", line 1
'Partirò domani all'alba'
^
SyntaxError: invalid syntax
```

Il problema è stato risolto definendo delle rappresentazioni simboliche sia dei caratteri di controllo sia di altri caratteri che per varie ragioni non possono essere inseriti facilmente nelle stringhe. Queste rappresentazioni sono dette *sequenze di escape*, e possono essere raggruppate in quattro categorie:

1. sequenze rappresentative di alcuni caratteri ASCII di controllo: `\n`, `\t` ecc.;
2. sequenze rappresentative di alcuni caratteri ASCII stampabili: `\\`, `\'`, `\"`;
3. sequenze che consentono di rappresentare una parte dei caratteri Unicode: `\x41`, `\101`;
4. sequenze che consentono di rappresentare qualunque carattere Unicode: `\u0041` ecc.

Le sequenze di escape iniziano con un carattere di *backslash*, indipendentemente dalla categoria, e rappresentano un unico carattere. Ad esempio `\n` non rappresenta una stringa composta dai caratteri singoli `\` e `n`,

ma è un solo carattere, precisamente il carattere ASCII che rappresenta una interruzione di linea (il carattere di *newline*, avente codice 10). Infatti, quando si calcola la lunghezza di una stringa, ogni sequenza di escape conta per un solo carattere:

```
>>> len('\n'), len('ab\n'), len(b'\n')
(1, 3, 1)
```

Se il carattere rappresentato dalla sequenza di escape non può essere visualizzato all'interno della stringa (ad esempio perché è un carattere di controllo e non ha rappresentazione grafica), allora la shell interattiva mostra la sequenza così come la scriviamo, senza interpretarne il significato. Ad esempio, quando la shell interattiva incontra un'interruzione di linea, la rappresenta con la sequenza di escape `\n` e non crea un'interruzione di linea:

```
>>> open('myfile').read() # La shell interattiva mostra i caratteri di escape senza interpretarli
'prima\nseconda'
```

e quando incontra, all'interno della stessa stringa, sia un apice singolo sia un doppio apice, può visualizzare uno dei due, ma è costretta a rappresentare l'altro con una sequenza di escape:

```
>>> "Gli ho detto: \"ti aiuto anche io a preparare l'albero di Natale!\""
'Gli ho detto: "ti aiuto anche io a preparare l\'albero di Natale!"'
```

La funzione built-in `print()`, invece, non mostra mai le sequenze di escape contenute all'interno delle stringhe di testo, poiché ne interpreta il significato e stampa il carattere a esse associato. Ad esempio, la sequenza `\n` viene interpretata come un'interruzione di linea e stampata come tale:

```
>>> s = open('myfile').read()
>>> s
'prima\nseconda'
>>> print(s)
prima
seconda
```

NOTA

La shell interattiva stampa il risultato delle espressioni tramite la funzione `sys.displayhook()`. In sostanza, se `obj` è l'oggetto ottenuto come risultato di una espressione, la shell interattiva chiama automaticamente `sys.displayhook(obj)`:

```
>>> s = 'prima\nseconda'
>>> import sys
>>> sys.displayhook(s)
'prima\nseconda'
```

La funzione `sys.displayhook()` non fa altro che stampare la rappresentazione dell'oggetto passato come argomento (se questo è diverso da `None`). La *rappresentazione di un oggetto* `obj` si ottiene chiamando la funzione built-in `repr(obj)`, la quale restituisce una rappresentazione di `obj` sotto forma di stringa. Quindi la chiamata `sys.displayhook(obj)` è analoga alla chiamata `print(repr(obj))`. Questo significa che, se vogliamo cambiare il modo in cui la shell interattiva mostra il risultato di una espressione, possiamo assegnare un'altra funzione a `sys.displayhook`:

```
>>> import sys
>>> displayhook_copy = sys.displayhook # Salvo una copia
>>> def mydisplayhook(obj):
...     print('Il risultato ->', obj)
...
>>> sys.displayhook = mydisplayhook
>>> 44
Il risultato -> 44
>>> [1, 2, 3]
Il risultato -> [1, 2, 3]
>>> sys.displayhook = displayhook_copy # Ripristino `sys.displayhook`
>>> 44
44
```

Il *display hook* è discusso nella PEP-0217 e nella documentazione online del modulo `sys`.

Quando viene mostrata una stringa di byte, i caratteri di escape non vengono mai interpretati. Come abbiamo già detto, infatti, le stringhe di byte sono sempre rappresentate mediante il loro letterale:

```
>>> print(s.encode())
b'prima\nseconda'
```

Sequenze di escape rappresentative di alcuni caratteri ASCII di controllo

Le sequenze di escape appartenenti a questa categoria sono `\n`, `\t`, `\v`, `\r`, `\b`, `\a` e `\f`, e consentono di rappresentare alcuni caratteri ASCII di controllo.

Come sappiamo, `\n` rappresenta il carattere di interruzione di linea, detto *newline*, corrispondente al carattere ASCII con codice 10 (code point U+000A).

Le sequenze di escape `\t` e `\v` servono per rappresentare i caratteri di tabulazione. La sequenza `\t` rappresenta un carattere di *tabulazione orizzontale*, avente codice ASCII 9 (code point U+0009), mentre `\v` rappresenta un carattere di *tabulazione verticale*, avente codice ASCII 11 (code point U+000B):

```
>>> ord('\t'), ord('\v')
(9, 11)
>>> s = 'python\t3'
>>> print(s)
python 3
>>> s = 'python\v3'
>>> print(s)
python
3
```

La sequenza di escape `\b` rappresenta il carattere ASCII di *backspace*, avente codice 8:

```
>>> ord('\b') # Code point U+0008
8
>>> s = 'python\b3'
>>> print(s)
pytho3
```

La sequenza `\r` rappresenta il carattere ASCII di *carriage return* (ritorno a capo), avente codice 13:

```
>>> ord('\r') # Code point U+000D
13
>>> s = 'python\r3'
>>> print(s)
3ython
```

Le ultime due sequenze di questa categoria sono `\a` e `\f`, e rappresentano, rispettivamente, il carattere ASCII *bell*, che provoca un segnale sonoro, e il carattere ASCII *form feed*, utilizzato per indicare una nuova pagina.

Per quanto riguarda le stringhe di byte, abbiamo detto che, quando il valore di un byte coincide con il codice di un carattere ASCII, la shell interattiva mostra il carattere piuttosto che la notazione `\xhh`. Quando il carattere ASCII in questione è un carattere di controllo, se vi è una sequenza di escape che lo rappresenta, viene mostrata in questo modo:

```
>>> b'\x00\x0A\x09'
b'\x00\n\t'
```

Sequenze di escape rappresentative di alcuni caratteri ASCII stampabili

Vi sono alcuni caratteri che talvolta non possono essere inseriti nelle stringhe. Si pensi, ad esempio, al carattere di apice singolo e di doppio apice. Quello di apice singolo non può essere inserito in un letterale stringa delimitato da apici singoli; la stessa cosa vale per il doppio apice in un letterale delimitato da doppi apici. Se non dobbiamo inserire contemporaneamente i due tipi di apice nello stesso letterale, il problema non si pone poiché è sufficiente invertire l'utilizzo degli apici, nel senso che, se dobbiamo inserire nella stringa un apice singolo, allora delimitiamo il letterale con doppi apici, e viceversa:

```
>>> "L'altro giorno ho visto Anna Chiara."  
"L'altro giorno ho visto Anna Chiara."  
>>> 'Le ho chiesto: "tutto bene?"'  
'Le ho chiesto: "tutto bene?"'
```

Se, però, volessimo racchiudere il testo delle precedenti due stringhe all'interno di un unico letterale, dovremmo fare ricorso alle sequenze di escape `\'` e `\"`. La prima rappresenta il carattere ASCII di *apice singolo* (codice 39, U+0027), la seconda il carattere ASCII di *doppio apice* (codice 34, U+0022):

```
>>> 'L\'altro giorno ho visto Anna Chiara. Le ho chiesto: "tutto bene?"'  
'L\'altro giorno ho visto Anna Chiara. Le ho chiesto: "tutto bene?"'  
>>> "L'altro giorno ho visto Anna Chiara. Le ho chiesto: \"tutto bene?\""  
'L'altro giorno ho visto Anna Chiara. Le ho chiesto: "tutto bene?"'
```

Quando una stringa di byte contiene sia un byte con valore corrispondente al codice di un apice singolo sia uno con valore corrispondente al codice di un doppio apice, uno dei due viene rappresentato mediante la corrispondente sequenza di escape:

```
>>> b'\x27\x22' # 0x27 è il codice di un apice singolo e 0x22 quello di un doppio apice  
b'\''
```

L'altra sequenza di escape appartenente a questa categoria è `\\`. Questa rappresenta il carattere di *backslash* (codice 92, U+005C), e viene utilizzata per evitare che il backslash venga considerato come l'inizio di una sequenza di escape. Proviamo a chiarire il concetto con un esempio. Supponiamo di lavorare con un sistema Unix-like e di voler leggere il contenuto dei file `out\myfile.data` e `out\newfile.data`:

```
>>> data = open('out\\myfile.data').read()  
>>> print(data, end='')
```

```
Contenuto del file `out\myfile.data`
>>> data = open('out\newfile.data').read()
Traceback (most recent call last):
...
FileNotFoundError: [Errno 2] No such file or directory: 'out\newfile.data'
```

Python dice che non trova *out\newfile.data*, ma in realtà il file esiste:

```
>>> import os
>>> for filename in os.listdir():
...     print(filename)
...
out\myfile.data
out\newfile.data
```

Il problema sta nel backslash contenuto nella stringa 'out\newfile.data'. Infatti, come sappiamo, '\n' è la sequenza di escape che rappresenta il carattere di newline. Il problema non si è manifestato con la stringa 'out\myfile.data' poiché '\m' non è una sequenza di escape, ma una stringa composta dai caratteri singoli \ e m:

```
>>> print('\m')
\m
>>> len('\m')
2
```

Vi sono due soluzioni per fare in modo che '\n' non venga considerata come una sequenza di escape, ma piuttosto come una stringa composta dai due caratteri \ e n. La prima soluzione consiste nel rappresentare il backslash mediante la sequenza di escape \\:

```
>>> for c in '\\n':
...     print(c)
...
\
n
>>> data = open('out\\newfile.data').read() # Questa volta infatti riusciamo ad aprire il file
>>> print(data)
Contenuto del file `out\newfile.data`
```

La seconda soluzione consiste nell'utilizzare il cosiddetto *letterale stringa raw*, che viene creato facendo precedere da una lettera r (indifferentemente minuscola o maiuscola) i letterali stringa che già conosciamo. I caratteri all'interno di una stringa raw, infatti, vengono sempre considerati singolarmente, e mai come parte di una sequenza di escape:

```
>>> len(r'\n')
2
>>> data = open(r'out\newfile.data').read()
>>> print(r'non sono\n due linee')
non sono\n due linee
```

Anche le stringhe di byte hanno un letterale raw:

```
>>> s = rb'\n'
>>> len(s)
2
>>> chr(s[0]), chr(s[1])
('\', 'n')
```

Sequenze di escape che consentono di rappresentare una parte dei caratteri Unicode

Le sequenze di escape appartenenti a questa categoria consentono di rappresentare una parte dei caratteri Unicode indicando il loro codice, espresso in formato ottale o esadecimale. Consideriamo, ad esempio, il carattere con codice decimale 200 (0xC8 in esadecimale e 0o310 in ottale):

```
>>> chr(200)
'È'
```

Questo può essere rappresentato tramite le seguenti sequenze di escape:

```
>>> '\xC8', '\310'
('È', 'È')
```

Nella rappresentazione del codice in formato ottale il massimo numero di cifre utilizzabili è tre, per cui si possono rappresentare al massimo 512 caratteri (codici da 0o0 a 0o777). Ecco alcuni esempi:

```
>>> '\120'
'P'
>>> '\63'
'3'
>>> '\120ython \63'
'Python 3'
>>> b'\143\151\141\157'
b'ciaio'
```

Nella rappresentazione del codice in formato esadecimale è possibile utilizzare al massimo due cifre, per cui si possono rappresentare non più di 256 caratteri (codici da 0x00 a 0xFF). Ecco un esempio di utilizzo:

```
>>> '\x50'
'P'
>>> '\x33'
'3'
>>> '\x50ython \x33'
'Python 3'
>>> b'\x63\x69\x61\x6f'
b'ciaof'
```

Sequenze di escape che consentono di rappresentare qualunque carattere Unicode

Le sequenze di escape appartenenti a questa categoria consentono di rappresentare nelle stringhe di testo tutti i caratteri Unicode indicando il loro nome oppure il loro code point.

Le sequenze di escape che utilizzano la *property name* del carattere vanno scritte con la sintassi `\N{property name}`, con `property name` scritto indifferentemente maiuscolo o minuscolo. Ad esempio, la *property name* del carattere A è `LATIN CAPITAL LETTER A`:

```
>>> import unicodedata
>>> unicodedata.name('A')
'LATIN CAPITAL LETTER A'
```

e quindi il carattere A può essere rappresentato con le seguenti sequenze di escape:

```
>>> '\N{LATIN CAPITAL LETTER A}'
'A'
>>> '\N{latin capital letter a}'
'A'
```

Le sequenze di escape che utilizzano i code point vanno scritte con la sintassi `\Uhhhhhhhh`, con otto cifre esadecimali:

```
>>> '\U00000041' # LATIN CAPITAL LETTER A
'A'
```

Esiste anche la sintassi `\uhhhh`. A differenza della precedente, la lettera `u` va indicata in minuscolo e il numero di cifre deve essere pari a quattro, non otto, per cui si possono rappresentare solo i caratteri con codici da `U+0000` a `U+FFFF` (in decimale da 0 a 65535):

```
>>> '\u0041'
'A'
```

Questo tipo di sequenza di escape non ha effetto se la si utilizza nelle stringhe di byte, anche se il code point è quello di un carattere ASCII:

```
>>> s = b'\u0041'  
>>> s, len(s)  
(b'\u0041', 6)
```

Metodi delle stringhe

In questa sezione parleremo dei principali metodi delle stringhe. Ordineremo il discorso tenendo conto del fatto che alcuni metodi sono comuni a entrambi i tipi `str` e `bytes`, mentre altri appartengono a uno solo dei due tipi.

Metodi comuni alle stringhe di testo e alle stringhe di byte

Utilizziamo un set per vedere quali sono i metodi comuni ai due tipi di stringa:

```
>>> smethods = [name for name in dir(str) if not name.startswith('__')]  
>>> bmethods = [name for name in dir(bytes) if not name.startswith('__')]  
>>> set(smethods) & set(bmethods)  
{'startswith', 'swapcase', 'isupper', 'upper', 'split', 'count', 'rpartition',  
'lstrip', 'find', 'rfind', 'lower', 'istitle', 'endswith', 'join', 'maketrans',  
'isdigit', 'index', 'title', 'isalpha', 'isalnum', 'rjust', 'translate',  
'expandtabs', 'rstrip', 'zfill', 'splitlines', 'center', 'capitalize', 'islower',  
'partition', 'rindex', 'ljust', 'replace', 'strip', 'rsplit', 'isspace'}
```

Visto che i metodi comuni si comportano in modo identico sia nel caso di stringhe di byte sia in quello di stringhe di testo, non ripeteremo gli esempi per entrambi i tipi.

Le stringhe sono oggetti immutabili, per cui i metodi che apparentemente le modificano in realtà restituiscono una nuova stringa:

```
>>> s = "sono una stringa di caratteri minuscoli"  
>>> s.capitalize() # Restituisce una copia di s con il primo carattere maiuscolo  
'Sono una stringa di caratteri minuscoli'  
>>> s # La stringa originale non cambia  
'sono una stringa di caratteri minuscoli'  
>>> s.replace("minuscoli", "lowercase")  
'sono una stringa di caratteri lowercase'  
>>> s # Neanche questa volta la stringa originale è cambiata  
'sono una stringa di caratteri minuscoli'
```

Il metodo `str.replace()` che abbiamo appena visto ha anche un terzo argomento da utilizzare per indicare il numero di sostituzioni che devono essere effettuate:

```
>>> s = 'sono una stringa, quindi una sequenza immutabile di caratteri'  
>>> s.replace('una', 'UNA')
```



```
'sono UNA stringa, quindi UNA sequenza immutabile di caratteri'
>>> s.replace('una', 'UNA', 1)
'sono UNA stringa, quindi una sequenza immutabile di caratteri'
>>> s.replace('t', 'T', 3)
'sono una sTringa, quindi una sequenza immuTabile di caraTteri'
```

Si ricordi che tra stringhe di testo e stringhe di byte non avvengono mai conversioni implicite, per cui non è possibile passare al metodo `str.replace()` argomenti di tipo `bytes` e, viceversa, non è possibile passare a `bytes.replace()` argomenti di tipo `str`:

```
>>> b'fooooo'.replace('foo', 'moo')
Traceback (most recent call last):
...
TypeError: expected bytes, bytearray or buffer compatible object
```

Quando si vogliono effettuare sostituzioni più complesse rispetto a quelle appena viste, i metodi `str.maketrans()` e `str.translate()`, usati assieme, sono più efficaci di `str.replace()`. Consideriamo, ad esempio, la stringa 'Programmare con Python' e supponiamo di voler sostituire le lettere `a` con `A`, le lettere `b` con `B` e le lettere `c` con `C`. Come prima cosa, dobbiamo creare una cosiddetta tabella di traslazione, chiamando il metodo `str.maketrans()`:

```
>>> transtab = str.maketrans('abc', 'ABC')
```

Come possiamo vedere, il primo argomento di `str.maketrans()` è la stringa contenente i caratteri da sostituire, mentre il secondo argomento è la stringa contenente i nuovi caratteri. Le due stringhe devono avere uguale lunghezza, perché ogni carattere della prima verrà sostituito con il corrispondente carattere della seconda. Il metodo `str.maketrans()` restituisce una cosiddetta *tabella di traduzione*, che non è altro che un dizionario avente per chiavi i numeri ordinali dei caratteri da sostituire e per valori i numeri ordinali dei corrispondenti sostituiti:

```
>>> transtab
{97: 65, 98: 66, 99: 67}
```

A questo punto, data la stringa da trasformare:

```
>>> s = 'Programmare con Python'
```

possiamo chiamare il metodo `str.translate()` passandogli la tabella di traduzione:

```
>>> s.translate(transtab)
'ProgrAmmAre Con Python'
```

Il metodo `str.maketrans()` può essere chiamato anche passandogli un dizionario avente per chiavi i numeri ordinali dei caratteri da sostituire e per valori i numeri ordinali dei corrispondenti caratteri sostituiti:

```
>>> transtab = str.maketrans({97: 65, 98: 66, 99: 67})
>>> s.translate(transtab)
'ProgrAmmAre Con Python'
```

I caratteri mappati con `None` vengono cancellati, come accade nel seguente caso per i caratteri `y` e `h`:

```
>>> transtab = str.maketrans({97: 65, 98: 66, 99: 67, ord('y'): None, ord('h'): None})
>>> s.translate(transtab)
'ProgrAmmAre Con Pton'
```

Il metodo `str.split()`, che già conosciamo, spezza la stringa in varie parti e restituisce una lista contenente queste ultime. Per default la stringa viene spezzata in corrispondenza degli spazi, ma è possibile passare al metodo una stringa che indica l'elemento in corrispondenza del quale effettuare la separazione:

```
>>> 'Un giorno, se avrò del tempo, andrò a correre'.split()
['Un', 'giorno,', 'se', 'avrò', 'del', 'tempo,', 'andrò', 'a', 'correre']
>>> 'Un giorno, se avrò del tempo, andrò a correre'.split(',')
['Un giorno', ' se avrò del tempo', ' andrò a correre']
>>> 'Un giorno, se avrò del tempo, andrò a correre'.split('rò')
['Un giorno, se av', ' del tempo, and', ' a correre']
```

Tramite l'argomento opzionale `maxsplit` è possibile indicare il massimo numero di volte in cui la stringa va spezzata:

```
>>> 'Un giorno, se avrò del tempo, andrò a correre'.split(maxsplit=3)
['Un', 'giorno,', 'se', 'avrò del tempo, andrò a correre']
```

Data una stringa `s`, il metodo `s.join()` prende come argomento un oggetto iterabile i cui elementi sono stringhe e restituisce una stringa composta dagli elementi dell'oggetto iterabile, separati da `s`:

```
>>> ''.join(['1', '2', 'tre'])
'12tre'
>>> '-'.join(['1', '2', 'tre'])
'1-2-tre'
```

```
>>> f = open('myfile', 'w')
>>> f.writelines(['prima\n', 'seconda\n', 'terza\n'])
>>> f.close()
>>> '-'.join(open('myfile'))
'prima\n-seconda\n-terza\n'
```

NOTA

I metodi `str.split()` e `str.join()` sono complementari:

```
>>> s = 'python3'
>>> 'n'.join(s.split('n'))
'python3'
```

Possiamo sfruttare questa caratteristica per aggiungere dei caratteri in prossimità di un separatore:

```
>>> 'n '.join(s.split('n'))
'python 3'
>>> 'n-'.join(s.split('n'))
'python-3'
```

I metodi `str.lstrip()`, `str.rstrip()`, `str.strip()` per default rimuovono da una stringa i caratteri di spaziatura, rispettivamente a sinistra (*left strip*), a destra (*right strip*) e sia a sinistra sia a destra:

```
>>> s = '\n\n\t stringa con newline \n'
>>> s.lstrip() # Elimina gli spazi alla sinistra (left) della stringa
'stringa con newline \n'
>>> s.rstrip() # Elimina gli spazi alla destra (right) della stringa
'\n\n\t stringa con newline'
>>> s.strip() # Elimina gli spazi all'inizio e alla fine della stringa
'stringa con newline'
```

Vediamo cosa accade se a uno di questi tre metodi si passa come argomento una stringa:

```
>>> 'stringa'.lstrip('atgrs')
'inga'
```

Come possiamo vedere, è stato prima eliminato il carattere `s` in quanto presente in `'atgrs'`. A questo punto da `'tringa'` viene rimosso il carattere iniziale `t`, poiché anch'esso è presente in `'atgrs'`. Ora da `'ringa'` viene rimosso il carattere iniziale `r`, poiché anch'esso è presente in `'atgrs'`. Dalla stringa `'inga'` non viene

rimossa la `i` poiché non è presente in `'atgrs'`, per cui l'operazione termina qui e il risultato è la stringa `'inga'`.

Il metodo `str.title()` restituisce una stringa che ha ogni iniziale di parola maiuscola:

```
>>> 'oggi è una bella giornata'.title() # Iniziali di parola maiuscole
'Oggi È Una Bella Giornata'
```

mentre il metodo `str.capitalize()` restituisce una stringa che inizia con la lettera maiuscola:

```
>>> 'oggi è una bella giornata'.capitalize()
'Oggi è una bella giornata'
```

Il metodo `str.upper()` restituisce una stringa con tutti i caratteri maiuscoli, mentre il metodo `str.lower()` ne restituisce una con tutti i caratteri minuscoli:

```
>>> 'oggi è una bella giornata'.upper() # Tutto in maiuscolo
'OGGI È UNA BELLA GIORNATA'
>>> 'Oggi è una BELLA giornata'.lower() # Tutto in minuscolo
'oggi è una bella giornata'
```

Vi sono poi dei metodi per effettuare vari test:

```
>>> 'oggi è una bella giornata'.startswith('oggi') # La stringa inizia con 'oggi'?
True
>>> 'oggi è una bella giornata'.endswith('ata') # La stringa finisce con 'ata'
True
>>> 'oggi è una bella giornata'.islower() # I caratteri della stringa sono tutti minuscoli?
True
>>> 'Oggi è una bella giornata'.isupper() # I caratteri della stringa sono tutti maiuscoli?
False
>>> 'OGGI È UNA BELLA GIORNATA'.isupper()
True
```

Il metodo `str.isalnum()` ci dice se la stringa è composta esclusivamente da caratteri alfanumerici:

```
>>> '122'.isalnum(), '122a'.isalnum()
(True, True)
```

Il metodo `str.isalpha()` ci dice se la stringa è composta da sole lettere dell'alfabeto latino:

```
>>> '122'.isalpha(), '122a'.isalpha(), 'abc!'.isalpha()
```

```
(False, False, False)
>>> '122'.isalpha(), '122a'.isalpha(), 'abc!'.isalpha(), 'abc'.isalpha()
(False, False, False, True)
```

Infine, vi è il metodo `str.isdigit()`, del quale vedremo il significato nella prossima sezione:

```
>>> '122'.isdigit(), '122a'.isdigit()
(True, False)
```

Metodi specifici delle stringhe di testo

Ecco l'elenco dei principali metodi specifici delle stringhe di testo:

```
>>> set(smethods) - set(bmethods)
{'isnumeric', 'isidentifier', 'isdecimal', 'isprintable', 'format_map', 'casefold', 'format', 'encode'}
```

Iniziamo con i metodi legati alle stringhe *decimali* e *numeriche*, mostrandovi questo esempio:

```
>>> '133'.isdigit(), '133'.isdecimal(), '133'.isnumeric()
(True, True, True)
```

Quindi tutti e tre i metodi restituiscono `True` quando la stringa contiene delle cifre. Ma cosa intendiamo, in realtà, con il termine *cifra*? Innanzitutto anticipiamo che questi metodi restituiscono `True` quando tutti i caratteri contenuti nella stringa appartengono alla categoria Unicode dei *Number, decimal digit*, indicata con *Nd*. Infatti per la stringa '133' si ha:

```
>>> import unicodedata
>>> for c in '133':
...     print(c, unicodedata.name(c), unicodedata.category(c), sep='\t')
...
1 DIGIT ONE Nd
3 DIGIT THREE Nd
3 DIGIT THREE Nd
```

Osserviamo, inoltre, che se una stringa contiene un `float`, i tre metodi restituiscono `False`:

```
>>> '1.33'.isdigit(), '1.33'.isdecimal(), '1.33'.isnumeric()
(False, False, False)
```

Infatti il carattere utilizzato per indicare il punto decimale non appartiene alla

categoria Nd:

```
>>> for c in '1.33':
...     print(c, unicodedata.name(c), unicodedata.category(c), sep='\t')
...
1 DIGIT ONE Nd
. FULL STOP Po
3 DIGIT THREE Nd
3 DIGIT THREE Nd
```

Per capire quale sia la differenza tra questi tre metodi, creiamo innanzitutto tre liste contenenti tutti i caratteri considerati, rispettivamente *decimal*, *digit* e *numeric*:

```
>>> import sys
>>> sys.maxunicode # Rappresenta il codice Unicode più grande
1114111
>>> decimals = [chr(c) for c in range(sys.maxunicode + 1) if chr(c).isdecimal()]
>>> digits = [chr(c) for c in range(sys.maxunicode + 1) if chr(c).isdigit()]
>>> numerics = [chr(c) for c in range(sys.maxunicode + 1) if chr(c).isnumeric()]
```

Possiamo constatare che non ci sono caratteri *decimal* che non siano *digit*, ma non vale il viceversa:

```
>>> set(decimals) - set(digits), len(set(digits) - set(decimals))
(set(), 128)
```

Quindi i *decimal* sono un sottoinsieme dei *digit*. Inoltre i *digit* sono un sottoinsieme dei *numeric*:

```
>>> set(digits) - set(numerics), len(set(numerics) - set(digits))
(set(), 637)
```

Quindi l'insieme dei caratteri considerati *numeric* contiene quello dei caratteri considerati *digit*, che a sua volta contiene quello dei caratteri considerati *decimal*.

Ora che abbiamo un'idea del legame fra i tre insiemi, vediamo di entrare più nel dettaglio. I caratteri Unicode che appartengono alla categoria *Nd* sono considerati *numeric*, *digit* o *decimal*, o una combinazione dei tre a seconda del valore della loro property *Numeric_Type*:

- se *Numeric_Type*=*Decimal*, tutti e tre i metodi restituiscono *True*;
- se *Numeric_Type*=*Digit*, allora solo i metodi *str.isdigit()* e *str.isnumeric()* restituiscono *True*;
- se i caratteri hanno *Numeric_Type*=*Numeric*, solo *str.isnumeric()* restituisce *True*.

NOTA

Il valore del *Numeric_Type* di un carattere Unicode viene assegnato sulla base dei seguenti criteri:

- *Numeric_Type*=Decimal se, nella riga dell'Unicode Data relativa al carattere in questione, i campi 6, 7 e 8 (colonne 7, 8 e 9) contengono un valore intero (compreso tra 0 e 9); l'intero contenuto in questi campi rappresenta il valore numerico assegnato al carattere. In altre parole, i caratteri con la property *Numeric_Type*=Decimal sono quelli che rappresentano delle cifre utilizzabili in un sistema di numerazione posizionale in base 10;
- *Numeric_Type*=Digit se i campi 7 e 8 contengono un valore intero (compreso tra 0 e 9) e il campo 6 è nullo;
- *Numeric_Type*=Numeric se il campo 9 contiene un valore numerico, positivo, negativo o razionale e i campi 6 e 7 sono nulli.

Ecco alcuni esempi di utilizzo dei tre metodi:

```
>>> '123'.isdecimal(), '123'.isdigit(), '123'.isnumeric()
(True, True, True)
>>> '2\u00B2'.isdecimal(), '2\u00B2'.isdigit(), '2\u00B2'.isnumeric()
('2²', False, True, True)
>>> '5\u2155'.isdecimal(), '5\u2155'.isdigit(), '5\u2155'.isnumeric()
('5½', False, False, True)
```

Il metodo `str.casefold()` viene utilizzato per normalizzare una stringa in modo da poter fare un confronto *case insensitive*:

```
>>> s1 = 'A presto Elisetta!'
>>> s2 = s1.title()
>>> s1, s2, s1 == s2
('A presto Elisetta!', 'A Presto Elisetta!', False)
>>> s1.casefold(), s2.casefold(), s1.casefold() == s2.casefold()
('a presto elisetta!', 'a presto elisetta!', True)
```

Se stiamo pensando che avremmo ottenuto lo stesso risultato utilizzando i metodi `str.upper()` o `str.lower()` piuttosto che `str.casefold()`, proviamo a convincervi che quella non è la strada giusta. Consideriamo, ad esempio, la parola tedesca *Grüßen*, utilizzata come forma di saluto. La lettera *ß* esiste solo in forma minuscola e viene resa maiuscola con *SS*, per cui:

```
>>> s1 = "Grüßen"
>>> s2 = s1.upper()
>>> s1, s2
```

```
('Grüßen', 'GRÜSSEN')
>>> s1.lower(), s2.lower(), s1.lower() == s2.lower()
('grüßen', 'grüssen', False)
>>> s1.casefold(), s2.casefold(), s1.casefold() == s2.casefold()
('grüssen', 'grüssen', True)
```

Parleremo dei metodi `str.format()` e `str.format_map()` nella sezione *Formattazione delle stringhe di testo*.

Metodi specifici delle stringhe di byte

Veniamo ai metodi specifici delle stringhe di byte:

```
>>> set(bmethods) - set(smehods)
{'fromhex', 'decode'}
```

Il metodo `bytes.decode()` lo conosciamo già, mentre il metodo `bytes.fromhex()` crea una stringa di byte a partire da una stringa di testo composta da caratteri rappresentativi di cifre esadecimali:

```
>>> bytes.fromhex('8AFF4C'), bytes.fromhex('8A FF 4C')
(b'\x8a\xffL', b'\x8a\xffL')
```

Formattazione delle stringhe di testo

La formattazione delle stringhe in Python si può eseguire in due modi: o mediante *espressioni di formattazione*, oppure, a partire da Python 2.6, mediante dei metodi preposti a fare ciò.

Espressioni di formattazione delle stringhe di testo

Questa prima tecnica di formattazione, ispirata alla `printf()` del linguaggio C, fa uso dell'operatore `%` che viene applicato nel seguente modo:

stringa da formattare % valori

Quando la stringa da formattare richiede più di un argomento, allora i valori da passare devono essere inseriti in una tupla esattamente in numero pari a quello dei valori richiesti. Vediamo qualche esempio:

```
>>> "Adoro %s" % 'Python' # Non è necessario usare una tupla
'Adoro Python'
>>> "%s %d è meraviglioso" % ('Python', 3) # Necessario usare una tupla
'Python 3 è meraviglioso'
```


Poiché le stringhe sono oggetti immutabili, la formattazione non le modifica ma ne restituisce di nuove. La generica struttura degli elementi che devono essere convertiti ha la seguente forma:

%[(chiave)][[opzioni]][[profondità]][[precisione]]tipo

L'unico parametro obbligatorio è il *tipo* di conversione, che può essere uno tra quelli indicati nella colonna di sinistra della Tabella 2.1.

Tabella 2.1 - Tipi di conversione disponibili per formattare le stringhe di testo.

Tipo	Significato
d	Intero in rappresentazione decimale, con segno
i	Intero in rappresentazione decimale, con segno
o	Intero in rappresentazione ottale, con segno
u	Identico a d (obsoleto)
x	Intero in rappresentazione esadecimale, con segno (caratteri minuscoli)
X	Intero in rappresentazione esadecimale, con segno (caratteri maiuscoli)
e	Floating point in formato esponenziale (carattere esponenziale minuscolo)
E	Floating point in formato esponenziale (carattere esponenziale maiuscolo)
f	Floating point in formato decimale
F	Floating point in formato decimale
g	Floating point: usa il formato esponenziale (minuscolo) se l'esponente è minore di -4 o non è minore della precisione, altrimenti usa il formato decimale
G	Floating point: usa il formato esponenziale (maiuscolo) se l'esponente è minore di -4 o non è minore della precisione, altrimenti usa il formato decimale
c	Carattere singolo (accetta interi o stringhe di un carattere)
r	Stringa (converte ogni oggetto utilizzando repr())
s	Stringa (converte ogni oggetto utilizzando str())
a	Stringa (converte ogni oggetto utilizzando ascii())

Ecco alcuni esempi:

```
>>> '%d | %o | %x | %X' % (12, 12, 12, 12)
'12 | 14 | c | C'
>>> '%e | %E' % (1.33532354, 1.33532354)
'1.335324e+00 | 1.335324E+00'
>>> '%g' % 0.000033532354
'3.35324e-05'
>>> '%g' % 0.00033532354
'0.000335324'
```

La *chiave* è utilizzata nella formattazione basata sui dizionari. Come possiamo vedere, viene inserita tra parentesi nella stringa da formattare e viene sostituita con il corrispondente valore.

```
>>> '%(name)s, %(pass)s, %(age)d' % {'age': 27, 'name': 'elise', 'pass': '2905'}
'elise, 2905, 27'
```

Se la chiave non esiste, viene sollevata una eccezione di tipo `KeyError`:

```
>>> '%(name)s, %(pasw)s, %(age)d' % {'age': 27, 'name': 'elise', 'pass': '2905'}
Traceback (most recent call last):
...
KeyError: 'pasw'
```

La *profondità* indica il numero minimo di caratteri da utilizzare per la conversione. I caratteri mancanti per default vengono riempiti con degli spazi vuoti:

```
>>> '%d' % 100
'100'
>>> '%6d' % 100 # Profondità pari a 6 caratteri
' 100'
>>> '%6s' % 'ciao' # Profondità pari a 6 caratteri
' ciao'
```

Quando la profondità viene indicata con `*`, allora il suo valore viene preso dal corrispondente elemento nella tupla:

```
>>> '%*s' % (8, 'ciao') # Profondità di 8
' ciao'
>>> '%*s' % (11, 100) # Profondità di 11
' 100'
>>> '%s ha %*d anni' % ('Carlo', 10, 30) # Profondità di 10
'Carlo ha 30 anni'
```

Il parametro *opzioni* è utilizzato per specificare eventuali opzioni di conversione. Ad esempio, quando questo parametro vale `0`, significa che i caratteri mancanti devono essere sostituiti con degli zeri:

```
>>> '%09d' % 100 # Profondità pari a 9 e riempimento con degli zeri
'000000100'
>>> '%0*d' % (9, 100) # Profondità pari a 9 e riempimento con degli zeri
'000000100'
>>> '%0*f' % (15, 100) # Profondità pari a 15 e riempimento con degli zeri
'00000100.000000'
```

Questa opzione è valida solo per le conversioni numeriche:

```
>>> '%0*s' %(9, 100) # Conversione non numerica: riempimento con spazi
' 100'
```

Quando opzioni vale -, significa che il testo deve essere *giustificato* a sinistra:

```
>>> '%15f' %100 # Profondità pari a 15, per default la giustificazione è a destra
' 100.000000'
>>> '%-15f' %100 # Profondità pari a 15 con giustificazione a sinistra
'100.000000 '
```

L'opzione - annulla il riempimento con zeri qualora questo fosse presente:

```
>>> '%015f' %100 # Profondità pari a 15 e riempimento con degli zeri
'00000100.000000'
>>> '%-015f' %100 # La giustificazione a sinistra annulla il riempimento con zeri
'100.000000 '
```

La profondità può essere utilizzata, ad esempio, per incolonnare dei dati in un file:

```
>>> for i in [1, 10, 100]:
...     f.write('%-7d%-7d%-7d\n' %(i, i**2, i**3))
...
22
22
22
>>> f.close()
>>> for line in open('myfile'):
...     print(line)
...
1 1 1
10 100 1000
100 10000 1000000
```

Se come opzione si usa uno *spazio*, significa che deve essere lasciato uno spazio davanti ai numeri positivi:

```
>>> >>> '% d' %100
' 100'
>>> '% d' %-100
'-100'
>>> '% f' %100
' 100.000000'
```

Con l'opzione + il numero viene fatto precedere dal segno:

```
>>> '%+d' %100
'+100'
```

L'opzione + annulla l'opzione di spazio qualora questa fosse presente.

```
>>> '%+ d' %100
'+100'
>>> '% +d' %100
'+100'
```

Il quarto parametro è la *precisione*. Questo consente di specificare il numero di cifre da usare dopo il punto decimale. Viene fatto un arrotondamento, non un troncamento:

```
>>> num = 1.123456789
>>> '%f | %8.2f | %8.4f' % (num, num, num)
'1.123457 | 1.12 | 1.1235'
```

Ovviamente ha senso usarlo solo con i float.

Formattazione delle stringhe di testo tramite metodi e funzioni built-in

È possibile formattare una stringa di testo anche tramite il metodo `str.format()`. Questo utilizza la stringa come un *template*, prende un numero arbitrario di argomenti (che rappresentano i valori da sostituire) e restituisce una nuova stringa. Gli elementi da sostituire sono racchiusi tra parentesi graffe e i loro valori vengono reperiti tra gli argomenti di `str.format()` in modo posizionale e/o per keyword:

```
>>> '{name}, {0}, e anche una lista: {mylist}'.format(0, mylist=[0, 1, 2], name='Marco')
'Marco, 0, e anche una lista: [0, 1, 2]'
```

L'esempio appena visto è equivalente al seguente codice che utilizza le espressioni di formattazione:

```
>>> '%s, %d, e anche una lista: %s' % ('Marco', 0, [1, 2, 3])
'Marco, 0, e anche una lista: [1, 2, 3]'
```

Gli argomenti non-keyword di `str.format()` vanno indicati prima di quelli keyword:

```
>>> '{name}', {1}, e anche una lista: {mylist}'.format(mylist=[0, 1, 2], 33, name='Marco')
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Se non si indica alcuna posizione, la numerazione avviene in modo automatico:

```
>>> '{} is better than {}'.format('simple', 'complex')
'simple is better than complex'
```

Si possono anche utilizzare le chiavi dei dizionari e gli attributi degli oggetti. Consideriamo, ad esempio, il dizionario `os.environ` e l'attributo `platform` del modulo `sys`:

```
>>> import sys, os
>>> "L'utente {d[USERNAME]} usa {d[SHELL]} su {0.platform}".format(sys, d=os.environ)
"L'utente marco usa /bin/bash su linux"
```

L'esempio appena visto è equivalente al seguente codice che utilizza le espressioni di formattazione:

```
>>> import sys, os
>>> "L'utente %s usa %s su %s" %(os.environ['USERNAME'], os.environ['SHELL'],
... sys.platform)
```

È anche possibile effettuare l'indicizzazione di una sequenza:

```
>>> '{names[0]} usa {0[1]}'.format(('C++', 'Python'), names=('Marco', 'Leo'))
'Marco usa Python'
```

In questo caso sono ammessi solamente gli indici positivi:

```
>>> '{names[-1]} usa {0[1]}'.format(('C++', 'Python'), names=('Marco', 'Leo'))
Traceback (most recent call last):
...
TypeError: tuple indices must be integers, not str
```

Gli elementi possono essere annidati:

```
>>> '{0:. {1}f}'.format(1/3, 2)
'0.33'
```

La forma generale dell'elemento da sostituire è la seguente:

```
{elemento!indicatore_di_conversione:indicatore_di_formattazione}
```

dove i vari parametri hanno il seguente significato:

- l'*elemento*, come abbiamo visto, può essere un numero (sostituzione posizionale) o il nome di un argomento di `str.format()`, e questi possono essere seguiti da `.nome_attributo`, da `[indice]` per le sequenze o da `[chiave]` per i dizionari;
- l'*indicatore di conversione* può essere `r`, `s` o `a`, a seconda che si voglia effettuare la conversione chiamando le funzioni built-in `repr()`, `str()` o `ascii()`;
- l'*indicatore di formattazione* stabilisce come l'elemento deve essere presentato, definendo profondità, allineamento, precisione, tipo ecc. La forma generale è la seguente:

```
[riempimento[allineamento]][segno][#][0][profondità][.precision][tipo]
```

Per maggiori dettagli sulla forma generale di formattazione, si veda la sezione *Standard Format Specifiers* della PEP-3101.

Con il metodo `str.format()`, a differenza delle espressioni di formattazione, si può convertire un numero intero anche in notazione binaria:

```
>>> '{0:d}, {0:x}, {0:o}, {0:b}'.format(15)
'15, f, 17, 1111'
```

Se si vuole ottenere l'output analogo a quello che si avrebbe con le funzioni built-in `hex()`, `oct()` e `bin()`, si utilizza il carattere `#`, il quale, inoltre, fa sì che per i tipi `float`, `complex` e `decimal.Decimal` venga sempre mostrato il punto decimale, anche quando non è seguito da alcuna cifra:

```
>>> '{0#x}, {0#o}, {0#b}'.format(15)
'0xf, 0o17, 0b1111'
>>> '{0#.0f}, {0:.0f}'.format(1000)
'1000., 1000'
```

Con il metodo `str.format()`, inoltre, è possibile visualizzare il separatore delle migliaia in modo da rendere l'output più leggibile (PEP-0378). I tipi supportati sono `int`, `float`, `complex` e `decimal.Decimal`:

```
>>> '{:,d}'.format(1000000)
'1,000,000'
>>> '{:,3f}'.format(1000000.00)
'1,000,000.000'
>>> '{:,f}'.format(10000.00 + 10000.00j)
'10,000.000000+10,000.000000j'
>>> from decimal import Decimal
>>> '{:,3f}'.format(Decimal(1000000.30))
```

```
'1,000,000.300'
```

Oltre al metodo `str.format()` è possibile utilizzare per la formattazione la funzione built-in `format()`, la quale si comporta allo stesso modo di `str.format()` ma è limitata a un solo elemento da formattare:

```
>>> format(1/3, '.2f')
'0.33'
```

NOTA

La funzione built-in `format()` non fa altro che chiamare il metodo `__format__()` dell'oggetto. Come vedremo, quest'ultimo consente di definire lo *specificatore di formato* per ogni tipo di oggetto. Ad esempio, il tipo `datetime.datetime` ha il proprio *specificatore di formato*:

```
>>> from datetime import datetime
>>> "Oggi è: {0:%a %b %d %H:%M:%S %Y}".format(datetime.now())
'Oggi è: Sat Jul 14 00:04:55 2012'
```

Concludiamo questa sezione parlando del metodo `str.format_map()`, il quale estende le capacità del metodo `str.format()` accettando ogni oggetto mappatura:

```
>>> 'Oggi {nome} compie {età} anni'.format_map({'nome': 'Laura', 'età': 24})
'Oggi Laura compie 24 anni'
```

Questo rende possibile l'utilizzo della formattazione delle stringhe con ogni oggetto Python che ha la stessa interfaccia dei dizionari:

```
>>> import shelve
>>> d = shelve.open('myfile.shl')
>>> 'Mi chiamo {nome} e vivo a {città}'.format_map(d)
'Mi chiamo Marco e vivo a Nuoro'
```

Per maggiori dettagli sui metodi preposti alla formattazione, si veda la PEP-3101.

Stringhe di byte mutabili

Il tipo `bytearray` consente di creare delle stringhe di byte mutabili. La classe `bytearray` va chiamata passandole o una stringa di testo con la relativa codifica, oppure una stringa di byte:

```
>>> bytearray('Marco', 'latin-1'), bytearray(b'Marco')
(bytearray(b'Marco'), bytearray(b'Marco'))
```

Analogamente alle stringhe di byte, quando si accede a un singolo elemento di un bytearray, viene restituito un `int`, mentre viene restituito un `bytearray` se si esegue lo slicing:

```
>>> ba = bytearray(b'python')
>>> ba[0]
112
>>> ba[0:1] # Lo slicing restituisce un bytearray
bytearray(b'P')
```

Se si vuole modificare un elemento di un `bytearray` mediante l'indicizzazione, è necessario assegnargli un `int`, mentre, se gli si vuole assegnare una stringa di byte, si può utilizzare lo slicing:

```
>>> ba[0] = b'P'
Traceback (most recent call last):
...
TypeError: an integer is required
>>> ba[0] = 80
>>> ba
bytearray(b'Python')
>>> ba[:] = b'python'.upper() # Slicing in scrittura
>>> ba
bytearray(b'PYTHON')
```

I `bytearray` hanno tutti i metodi delle stringhe di byte e in più quelli tipici delle sequenze mutabili:

```
>>> bmethods = [name for name in dir(bytes) if not name.startswith('__')]
>>> amethods = [name for name in dir(bytearray) if not name.startswith('__')]
>>> set(bmethods) - set(amethods)
set()
>>> set(amethods) - set(bmethods)
{'remove', 'clear', 'append', 'reverse', 'copy', 'extend', 'insert', 'pop'}
```

I metodi che possono aggiungere o rimuovere un solo elemento prendono come argomento un `int`:

```
>>> ba = bytearray(b'PYTHON')
>>> ba.append(81) # Aggiunge l'elemento in coda al bytearray
>>> ba
bytearray(b'PYTHONQ')
>>> ba.remove(81) # Rimuove la prima occorrenza di 81
>>> ba
bytearray(b'PYTHON')
```



```
>>> ba.insert(6, 81) # Inserisce il valore 81 in sesta posizione
>>> ba
bytearray(b'PYTHONQ')
>>> ba.pop() # Restituisce e rimuove l'ultimo elemento
81
>>> ba
bytearray(b'PYTHON')
>>> ba.reverse() # Fa il reverse del bytearray
>>> ba
bytearray(b'NOHTYP')
>>> ba.extend(b'abc') # Estende il bytearray
>>> ba
bytearray(b'NOHTYPabc')
```

Concludiamo osservando che è possibile concatenare bytes con bytearray. Il risultato è una stringa di byte:

```
>>> b'python' + ba # È possibile concatenare bytes e bytearrays
b'pythonNOHTYPabc'
```

Liste

Le *liste* possono essere create, oltre che con il letterale, anche tramite la classe `list`, e, come sappiamo, anche con le *list comprehension*:

```
>>> list() # Crea una lista vuota
[]
>>> list((1, 2, 'tre')) # Può prendere un oggetto iterable
[1, 2, 'tre']
>>> list('python')
['p', 'y', 't', 'h', 'o', 'n']
>>> [c for c in 'Python']
['P', 'y', 't', 'h', 'o', 'n']
>>> [format(i, 'b') for i in range(10) if i % 2 == 0]
['0', '10', '100', '110', '1000']
```

NOTA

La creazione di una lista tramite list comprehension è più efficiente dell'equivalente codice con chiamata al metodo `list.append()`:

```
$ python -m timeit "[i for i in range(100)]"
100000 loops, best of 3: 5.28 usec per loop
$ python -m timeit "l = []" "for i in range(100): l.append(i)"
100000 loops, best of 3: 11.4 usec per loop
```

Le list comprehension possono anche essere annidate:

```
>>> matrix = [[1, 2, 3], [4, 5, 6]]
>>> transpose = [[row[i] for row in matrix] for i in range(3)]
>>> transpose
[[1, 4], [2, 5], [3, 6]]
```

Il codice equivalente è il seguente:

```
>>> transpose = []
>>> for i in range(3):
...     transpose.append([row[i] for row in matrix])
...
>>> transpose
[[1, 4], [2, 5], [3, 6]]
```

Metodi delle liste

I principali metodi delle liste sono i seguenti:

```
>>> [name for name in dir(list) if not name.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Il metodo `list.append()`, come ormai ben sappiamo, aggiunge un elemento in fondo alla lista. Il metodo `list.remove()` rimuove la prima occorrenza di un elemento dalla lista:

```
>>> mylist = [1, 2, 3, 1, 1]
>>> mylist.remove(1)
>>> mylist
[2, 3, 1, 1]
```

Se l'elemento non esiste, solleva una eccezione di tipo `ValueError`:

```
>>> mylist.remove(4)
Traceback (most recent call last):
...
ValueError: list.remove(x): x not in list
```

Il metodo `list.clear()` svuota la lista:

```
>>> mylist.clear()
>>> mylist
[]
>>> mylist = [1, 2, 3]
>>> del mylist[:] # Equivalente a mylist.clear()
>>> mylist
[]
```

Il metodo `list.copy()` effettua una *copia shallow* della lista originale, ovvero una copia i cui elementi sono dei riferimenti ai corrispondenti elementi della lista originale:

```
>>> l1 = [1, 2, ['a', 'b']]
>>> l2 = l1.copy() # Copia shallow
>>> l1 is l2
False
```

NOTA

I metodi `list.clear()` e `list.copy()` sono stati introdotti con Python 3.3.

Come sappiamo, si può fare una copia shallow anche con lo slicing:

```
>>> l3 = l1[:] # Copia shallow
>>> l1 is l3
False
>>> l1[1] is l2[1]
True
>>> l1[2] is l3[2]
True
>>> l1[2][1] = 'due'
>>> l1, l2, l3
([1, 2, ['a', 'due']], [1, 2, ['a', 'due']], [1, 2, ['a', 'due']])
>>> l1[2] = [1, 2] # Viene creato un nuovo oggetto
>>> l1, l2, l3
([1, 2, [1, 2]], [1, 2, ['a', 'due']], [1, 2, ['a', 'due']])
```

Abbiamo già visto, nella sezione dedicata ai dizionari, che per effettuare una *copia profonda* (*deepcopy*) di un oggetto si utilizza la funzione `deepcopy()` del modulo `copy`:

```
>>> l1 = [1, 2, ['a', 'b']]
>>> import copy
>>> l2 = copy.deepcopy(l1)
>>> l1 is l2
False
>>> l1[2] is l2[2]
False
>>> l1[2][1] = 'due'
>>> l1, l2
([1, 2, ['a', 'due']], [1, 2, ['a', 'b']])
```

Il metodo `list.pop()` restituisce e rimuove un elemento dalla lista. Se viene chiamato senza argomenti, rimuove e restituisce l'ultimo elemento:

```
>>> mylist = [1, 2, 3]
>>> mylist.pop()
3
>>> mylist
[1, 2]
```

Altrimenti, se gli si passa un indice, restituisce e rimuove l'elemento corrispondente:

```
>>> mylist.pop(0)
1
>>> mylist
[2]
```

Se l'indice è *out of range*, viene sollevata una eccezione di tipo `IndexError`:

```
>>> mylist.pop(2)
Traceback (most recent call last):
...
IndexError: pop index out of range
```

Il metodo `list.extend()` prende come argomento un oggetto iterabile e aggiunge i suoi elementi in coda alla lista:

```
>>> mylist = [1, 2, 3]
>>> mylist.extend({4, 5})
>>> mylist
[1, 2, 3, 4, 5]
```

Possiamo usarlo, quindi, per concatenare delle liste annidate in una lista:

```
>>> mylist = []
>>> for lst in [[1, 2, 3], ['a', 'b']]:
...     mylist.extend(lst)
...
>>> mylist
[1, 2, 3, 'a', 'b']
```

NOTA

Per concatenare delle liste annidate all'interno di una lista, tipicamente si utilizza la funzione `itertools.chain.from_iterable()`, che risulta la più efficiente:

```
>>> import itertools
>>> mylist = list(itertools.chain.from_iterable([[1, 2, 3], ['a', 'b']]))
>>> mylist
[1, 2, 3, 'a', 'b']
```

Un altro modo alternativo consiste nell'utilizzare l'augmented assignment:

```
>>> mylist = []
>>> for lst in [[1, 2, 3], ['a', 'b']]:
...     mylist += lst
...
>>> mylist
[1, 2, 3, 'a', 'b']
```

Ricordiamoci che dobbiamo evitare di utilizzare `sum()`, come abbiamo visto nella sezione *Il modulo itertools della libreria standard*.

Il metodo `list.index()` per default restituisce l'indice della prima occorrenza del suo argomento:

```
>>> ['a', 100, 33, 100].index(100)
1
```

Accetta gli argomenti opzionali `start` e `stop`, che indicano, rispettivamente, l'indice a partire dal quale iniziare la ricerca e quello in corrispondenza del quale terminarla:

```
>>> ['a', 'b', 1, 2, 3, 4, 'b', 'c'].index('b', 2)
6
>>> ['a', 'b', 1, 2, 3, 4, 'b', 'c'].index('b', 2, 7)
6
```

Se l'elemento cercato non esiste, viene sollevata una eccezione di tipo `ValueError`:

```
>>> ['a', 'b', 1, 2, 3, 4, 'b', 'c'].index('b', 2, 6)
Traceback (most recent call last):
...
ValueError: 'b' is not in list
```

Il metodo `list.count()` restituisce il numero di occorrenze di un elemento:

```
>>> ['a', 'b', 1, 2, 3, 4, 'b', 'c'].count('b')
2
```

Il metodo `list.reverse()` inverte gli elementi della lista:

```
>>> mylist = [1, 2, 3, 4]
>>> mylist.reverse() # Non restituisce nulla, ma modifica la lista
>>> mylist
```

```
[4, 3, 2, 1]
```

Il metodo `list.sort()` modifica la lista mettendo in ordine i suoi elementi. Il tipo di ordine viene stabilito dall'argomento opzionale `reverse`. Quando `reverse=False`, come nel caso di default, la lista viene ordinata disponendo gli elementi dal più piccolo al più grande, altrimenti dal più grande al più piccolo:

```
>>> mylist = [-4, 2, -5, 1, 7]
>>> mylist.sort()
>>> mylist
[-5, -4, 1, 2, 7]
>>> mylist.sort(reverse=True)
>>> mylist
[7, 2, 1, -4, -5]
```

Il metodo `list.sort()` accetta un argomento opzionale `key` che è analogo all'argomento `key` della funzione built-in `sorted()`:

```
>>> mylist.sort(key=abs)
>>> mylist
[1, 2, -4, -5, 7]
```

Se `list.sort()` non è in grado di ordinare gli elementi, lo segnala sollevando una eccezione:

```
>>> mylist = ['tre', 1, 2]
>>> mylist.sort()
Traceback (most recent call last):
...
TypeError: unorderable types: str() < int()
```

NOTA

In Python 2, a differenza di quanto avviene in Python 3, il metodo `list.sort()` può ordinare una lista contenente sia numeri sia stringhe:

```
>>> mylist = ['tre', 1, 2] # Python 2
>>> mylist.sort()
>>> mylist
[1, 2, 'tre']
```

Tuple

Le *tuple* possono essere create, oltre che con il letterale, anche tramite la classe `tuple`:

```
>>> tuple() # Senza argomenti crea una tupla vuota
()
>>> tuple([1, 2, 3]) # Può prendere come argomento un qualsiasi oggetto iterabile
(1, 2, 3)
>>> tuple('ciao')
('c', 'i', 'a', 'o')
```

Le parentesi tonde possono essere omesse quando la tupla viene creata sulla destra di un assegnamento:

```
>>> t = 1, 2, 3, 'quattro' # È possibile omettere le parentesi tonde in questo caso
>>> type(t)
<class 'tuple'>
```

Per creare una tupla di un solo elemento, è necessario utilizzare la virgola:

```
>>> >>> t = ('python') # Equivalente a `t = 'python'`
>>> type(t)
<class 'str'>
>>> t = ('python',)
>>> type(t)
<class 'tuple'>
```

Poiché le tuple sono oggetti immutabili, la loro lunghezza non cambia, l'indicizzazione e lo slicing non possono comparire alla sinistra dell'operatore di assegnamento e non hanno metodi che le modificano.

Le tuple hanno i metodi `tuple.index()` e `tuple.count()`:

```
>>> [name for name in dir(tuple) if not name.startswith('__')]
['count', 'index']
```

Questi si comportano in modo identico ai metodi `list.index()` e `list.count()`:

```
>>> (1, 2, 3).index(3) # Indice della prima occorrenza di 2
2
>>> (2, 4, 2, 5, 2).index(2, 3) # Prima occorrenza di 2 partendo dall'offset 3 (compreso)
4
>>> (1, 2, 3, 2, 2).count(2)
3
```

Le tuple con nome

La classe `namedtuple` del modulo `collection` della libreria standard consente di creare delle particolari tuple nelle quali ogni posizione ha un nome. Queste particolari tuple sono chiamate *tupla con nome* (*named tuple*). Vediamo di capire con un esempio di cosa si tratta. Creiamo una tupla con nome i cui

elementi rappresentano le coordinate di un punto nello spazio:

```
>>> from collections import namedtuple
>>> Point = namedtuple('MyPoint', ['x', 'y', 'z'])
>>> p = Point(10, 20, 30)
>>> p
MyPoint(x=10, y=20, z=30)
>>> p._fields
('x', 'y', 'z')
>>> p.x, p.y, p.z
(10, 20, 30)
>>> p[0], p[1], p[2]
(10, 20, 30)
>>> p.__qualname__
'MyPoint'
```

I nomi degli elementi della tupla possono essere inseriti in una stringa piuttosto che in una lista, separati da uno spazio o da una virgola:

```
>>> Point = namedtuple('MyPoint', 'x y z')
>>> p = Point(10, 20, 30)
>>> p.y
20
>>> Point = namedtuple('MyPoint', 'x, y, z')
>>> p = Point(10, 20, 30)
>>> p.z
30
```

La concatenazione tra una tupla e una tupla con nome restituisce una tupla:

```
>>> p + (1, 2, 3)
(10, 20, 30, 1, 2, 3)
```

Il metodo `namedtuple._asdict()` restituisce un `OrderedDict` avente per chiavi i nomi degli elementi e per valori gli elementi stessi:

```
>>> p._asdict()
OrderedDict([('x', 10), ('y', 20), ('z', 30)])
>>> p._asdict()['y']
20
```

Il metodo `namedtuple._replace()` restituisce una copia modificata della *named tuple* originale:

```
>>> new_p = p._replace(x=100)
>>> new_p
MyPoint(x=100, y=20, z=30)
>>> new_p = p._replace(x=1, y=2)
>>> new_p
MyPoint(x=1, y=2, z=30)
```


MyPoint(x=1, y=2, z=30)

Esercizio conclusivo

In questo esercizio conclusivo vedremo un programma che processa un file di log contenente dei dati di accesso a un sito web. Il programma in questione è il file *webstats.py*:

```
"""Processa un file di log contenente dei dati di accesso ad un sito web.
```

Le linee del file da processare devono essere di questo tipo:

```
66.249.73.89 - [29/Dec/2012:00:01:06 +0100]
```

contenenti quindi un indirizzo IP e la relativa data di accesso al sito.

Un esempio di utilizzo:

```
$ python webstats.py -f django_pariglias_com.log -d '2012 12 31'
```

In questo esempio lo script processa il file ``django_pariglias_com.log`` e mostra delle statistiche relative agli accessi al sito web avvenuti il 31 dicembre 2012.

```
"""
```

```
import sys
import argparse
import collections
from datetime import datetime, date, timedelta
```

```
def str2date(s):
```

```
    """Prendo una stringa 'year month day' e restituisco un `datetime.date`.
```

Un esempio di utilizzo:

```
>>> str2date('2013 1 16') # Nell'ordine: 'anno mese giorno'
```

```
datetime.date(2013, 1, 16)
```

```
"""
```

```
return date(*[int(item) for item in s.split()])
```

```
parser = argparse.ArgumentParser(
    description="Processa un file avente i dati di accesso ad un sito web")
parser.add_argument('-f', '--file', type=argparse.FileType(), required=True,
    help='File da processare')
parser.add_argument('-d', '--date', type=str2date, required=True,
    help="Data: 'y m d'")
parser.add_argument('-r', '--range', default=0, type=int, choices=range(3))
args = parser.parse_args()
```

```

total_hits = []

format_ = '%d/%b/%Y:%H:%M:%S %z' # day/month/year:hour:minute:second time-zone
for line in args.file:
    ip, hit_time = line.split(' - ')
    hit_date = datetime.strptime(hit_time.strip(), format_)
    if abs(args.date - hit_date.date()) <= timedelta(days=args.range):
        total_hits.append(ip) # Aggiungi l'indirizzo IP alla lista

unique_visitors = set(total_hits)
counter = collections.Counter()
for ip_address in total_hits:
    counter[ip_address] += 1 # Incrementa di una unità gli hits relativi a un IP

print('+ ' * 90)
print(args.file.name, *[args.date + timedelta(i)
    for i in range(-args.range, args.range + 1)], sep=' | ')
print('+ ' * 90)
print('Numero di hit:', len(total_hits)) if total_hits else sys.exit()
print('Visitatori unici: ', len(unique_visitors))
print('Visitatori che hanno effettuato il maggior numero di hit:')
for ip, hits in counter.most_common(5):
    print('\t%s -> %d' %(ip, hits))

```

Analizzeremo il tutto nelle sezioni che seguono; per il momento leggiamo ancora lo script e sforziamoci di capirne il significato in modo autonomo.

Esecuzione dello script

Iniziamo subito con un esempio di utilizzo:

```

$ ./webstats.py -f django_pariglias_com.log -d '2012 12 31'
+++++
django_pariglias_com.log | 2012-12-31
+++++
Numero di hit: 7051
Visitatori unici: 256
Visitatori che hanno effettuato il maggior numero di hit:
151.37.14.59 -> 641
82.53.27.146 -> 509
66.249.75.12 -> 417
94.39.232.38 -> 394
151.56.4.241 -> 298

```

Gli switch `-f` e `-d` precedono, rispettivamente, il nome del file e il giorno (*day*) del quale si vogliono calcolare le statistiche, per cui in questo esempio lo script legge il file *django_pariglias_com.log* e mostra a video delle statistiche di accesso relative al giorno 31 dicembre 2012.

I file di log che lo script è in grado di processare sono analoghi al seguente:

```
$ head -n 5 django_pariglias_com.log # Mostra le prime 5 linee del file
66.249.73.89 - [29/Dec/2012:00:01:06 +0100]
66.249.73.89 - [29/Dec/2012:00:01:11 +0100]
69.11.29.112 - [29/Dec/2012:00:02:57 +0100]
65.55.21.198 - [29/Dec/2012:00:03:01 +0100]
65.55.21.198 - [29/Dec/2012:00:03:03 +0100]
```

Ciascuna linea del file deve contenere i dati di un singolo hit, nella forma che abbiamo appena visto.

Lo script accetta un terzo argomento opzionale associato agli switch `-r` o `--range`. Questo argomento serve per indicare un *range* di giorni attorno a quello centrale specificato tramite `-d`. Ad esempio, se ci interessano le statistiche del 31 dicembre 2012 e dei due giorni precedenti e successivi a tale data, dobbiamo impostare un *range* pari a 2:

```
$ ./webstats.py -f django_pariglias_com.log -d '2012 12 31' -r 2
+++++
django_pariglias_com.log | 2012-12-29 | 2012-12-30 | 2012-12-31 | 2013-01-01 | 2013-01-02
+++++
Numero di hit: 31582
Visitatori unici: 932
Visitatori che hanno effettuato il maggior numero di hit:
78.15.201.37 -> 1756
93.70.137.20 -> 1152
95.238.93.85 -> 1030
79.5.153.196 -> 938
78.14.69.200 -> 920
```

Per default si ha `-r=0`, per cui vengono elaborate le statistiche per il solo giorno associato allo switch `-d`. Se non passiamo allo script gli argomenti obbligatori, viene mostrato un messaggio di errore:

```
$ ./webstats.py -d '2012 12 31'
usage: webstats.py [-h] -f FILE -d TARGET_DATE [-r {0,1,2}]
webstats.py: error: the following arguments are required: -f/--file
```

Ora che sappiamo ciò che fa lo script, vediamo di capire il significato del suo codice. Il primo passo consiste nello scoprire come lo script effettua il *parsing* degli argomenti passati da linea di comando.

Parsing degli argomenti passati da linea di comando

La parte iniziale dello script *webstats.py* si occupa di effettuare il parsing degli argomenti passati da linea di comando che, come sappiamo, sono contenuti nella lista `sys.argv`. Piuttosto che scrivere di nostro pugno del codice che fa il parsing degli argomenti di `sys.argv`, abbiamo giustamente scelto di utilizzare un

modulo che fa questo per noi: il modulo `argparse` della libreria standard. Le linee di codice che effettuano il parsing sono le seguenti:

```
parser = argparse.ArgumentParser(
    description="Processa un file avente i dati di accesso ad un sito web")
parser.add_argument('-f', '--file', type=argparse.FileType(), required=True,
    help='File da processare')
parser.add_argument('-d', '--date', type=str2date, required=True,
    help="Data: 'y m d'")
parser.add_argument('-r', '--range', default=0, type=int, choices=range(3))
args = parser.parse_args()
```

NOTA

Vi sono vari moduli che consentono di effettuare il parsing degli argomenti passati da linea di comando. In questo esercizio (e nel resto del libro) abbiamo usato `argparse` più che altro per comodità, poiché è disponibile con la libreria standard e quindi non necessita di alcuna installazione. Il modulo più “pythonico”, però, che probabilmente in futuro rimpiazzerà `argparse`, è `docopt` (<http://docopt.org/>). Questo probabilmente farà parte della libreria standard a partire da Python 3.5.

Il parser

L'oggetto che consente di effettuare il parsing si chiama *parser*. Nel nostro script il parser è una istanza della classe `ArgumentParser` del modulo `argparse`:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> type(parser)
<class 'argparse.ArgumentParser'>
```

La classe `ArgumentParser` accetta un argomento opzionale `description` che serve per indicare ciò che fa il programma. Questo argomento, se presente, viene assegnato all'attributo `description` del parser:

```
>>> parser = argparse.ArgumentParser(description="Descrizione del programma")
>>> parser.description
'Descrizione del programma'
```

Il namespace degli argomenti

Gli argomenti per i quali vogliamo che venga effettuato il parsing vanno dichiarati utilizzando il metodo `ArgumentParser.add_argument()`. Questo metodo si

occupa di analizzare gli argomenti dichiarati, dopodiché, se non rileva errori, restituisce un oggetto di tipo `argparse.Namespace`, che chiameremo *namespace degli argomenti*. Vediamo di capire ciò di cui stiamo parlando con un esempio. Consideriamo, a tale proposito, il seguente script:

```
$ cat argpar1.py
import argparse
parser = argparse.ArgumentParser() # Creiamo il parser
parser.add_argument('file') # Vogliamo fare il parsing di un argomento 'file'
args = parser.parse_args() # Parsing più assegnamento del namespace a args
lines = open(args.file).readlines() # Assegniamo a 'lines' la lista delle linee del file
print(lines[0]) # Stampiamo la prima linea del file
```

Con l'istruzione `parser.add_argument('file')` informiamo il parser che il programma dovrà prendere un argomento da linea di comando, che vogliamo venga assegnato all'attributo `file` del namespace. L'istruzione `args = parser.parse_args()` assegna all'etichetta `args` il namespace, per cui `args.file` è il nome del file passato da linea di comando. Quindi viene aperto il file di nome `args.file` e la lista delle sue linee viene assegnata all'etichetta `lines`. Infine, viene stampata la prima linea del file:

```
$ python argpar1.py django_pariglias_com.log
66.249.73.89 - [29/Dec/2012:00:01:06 +0100]
```

Se non passiamo al programma alcun argomento, il parser ci avviserà con un messaggio di errore:

```
$ python argpar1.py
usage: argpar1.py [-h] file
argpar1.py: error: the following arguments are required: file
```

Questo messaggio, oltre a mostrare l'errore, ci informa sul modo in cui il programma va utilizzato. Come possiamo vedere, è possibile utilizzare lo switch `-h`. Infatti il parser ha creato in modo automatico un help:

```
$ python argpar1.py -h
usage: argpar1.py [-h] file
```

```
positional arguments:
file
```

```
optional arguments:
-h, --help show this help message and exit
```

Affinché l'help sia veramente utile, dovrebbe mostrare un breve messaggio

che spieghi il significato dell'argomento `file`. Possiamo fare ciò passando il messaggio al metodo `parser.add_argument()`, nel modo seguente:

```
$ cat argpar2.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('file', help='Nome del file da aprire')
args = parser.parse_args()
lines = open(args.file).readlines()
print(lines[0])
```

```
$ python argpar2.py -h
usage: argpar2.py [-h] file
```

positional arguments:

file Nome del file da aprire

optional arguments:

-h, --help show this help message and exit

Il tipo degli argomenti

Supponiamo adesso di voler passare un secondo argomento da linea di comando, che identifica l'indice di una linea del file:

```
$ cat argpar3.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('file', help='Nome del file da aprire')
parser.add_argument('index', help='Indice della linea del file da visualizzare')
args = parser.parse_args()
lines = open(args.file).readlines()
print(lines[args.index])
```

```
$ python argpar3.py django_pariglias_com.log 2
```

```
Traceback (most recent call last):
```

```
...
```

```
TypeError: list indices must be integers, not str
```

Gli argomenti sono stringhe di testo, per cui `args.index` deve essere convertito in intero prima di effettuare l'indicizzazione. Potremmo risolvere il problema con `print(int(lines[args.index]))`, ma esiste una soluzione migliore. Possiamo passare a `parser.add_argument()` l'argomento `type=int`, e in questo modo `parser.parse_args()` effettuerà per noi la conversione, assegnando `args.index = int(sys.argv[2])`:

```
$ cat argpar4.py
import argparse
parser = argparse.ArgumentParser()
```

```

parser.add_argument('file', help='Nome del file da aprire')
parser.add_argument('index', type=int, help='Indice della linea del file da visualizzare')
args = parser.parse_args()
lines = open(args.file).readlines()
print(lines[args.index])

```

```

$ python argpar4.py django_pariglias_com.log 2
66.249.73.89 - [29/Dec/2012:00:01:43 +0100]

```

Al parametro `type` può essere assegnato un qualsiasi oggetto *callable* che prende una stringa come argomento e restituisce un oggetto convertito. Questo significa che, se passiamo al parametro `type` la funzione built-in `open()`, il parser effettua l'assegnamento `args.file = open('django_pariglias_com.log')`, per cui non dobbiamo neppure preoccuparci di aprire il file:

```

$ cat argpar5.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('file', type=open, help='Nome del file da aprire')
parser.add_argument('index', type=int, help='Indice della linea del file da visualizzare')
args = parser.parse_args()
lines = args.file.readlines() # Non abbiamo bisogno di aprire il file
print(lines[args.index])

```

```

$ python argpar5.py django_pariglias_com.log 2
66.249.73.89 - [29/Dec/2012:00:01:43 +0100]

```

Questo approccio presenta due problemi. Il primo consiste nel fatto che, se il file non esiste, il programma viene interrotto in modo poco piacevole:

```

$ python argpar5.py foofile 2
Traceback (most recent call last):
...
FileNotFoundError: [Errno 2] No such file or directory: 'foofile'

```

Il secondo, ancora più importante del primo, è dato dal fatto che, se volessimo aprire il file in scrittura, non potremmo farlo. Infatti abbiamo detto che il parametro `type` accetta un oggetto chiamabile che prende un solo argomento, per cui non abbiamo possibilità di passare a `open()` la modalità di apertura, e il file verrà aperto sempre con la modalità di default `mode='r'`. Per risolvere questo problema è stata definita nel modulo `argparse` la classe `argparse.FileType`, la quale prende un argomento `mode` che indica la modalità di apertura:

```

>>> import argparse
>>> file_type = argparse.FileType('r') # Modalità lettura

```



```
>>> file_type = argparse.FileType('w') # Modalità scrittura
>>> file_type = argparse.FileType() # Il valore di default è 'r'
>>> file_type._mode
'r'
```

L'oggetto restituito da `argparse.FileType` è una classe che prende il nome di un file come argomento e restituisce il corrispondente file-object. Grazie a questa strategia, possiamo quindi aprire il file anche in scrittura:

```
>>> file_type = argparse.FileType('w')
>>> file = file_type('foofile') # Apre il file 'foofile' in scrittura
>>> file.write('ciao')
4
>>> file.close()
>>> open('foofile').read()
'ciao'
```

Quindi, quando nello script *webstats.py* passiamo `type=argparse.FileType()`, il parser esegue delle azioni paragonabili alle seguenti:

```
>>> type_ = argparse.FileType()
>>> args = argparse.Namespace()
>>> args.file = type_('django_pariglias_com.log')
```

Aggiorniamo, quindi, il nostro script *argpar5.py* con una nuova versione:

```
$ cat argpar6.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('file', type=argparse.FileType(), help='Nome del file da aprire')
parser.add_argument('index', type=int,
help='Indice della linea del file da visualizzare')
args = parser.parse_args()
lines = args.file.readlines()
print(lines[args.index])
$ python argpar6.py django_pariglias_com.log 2
66.249.73.89 - [29/Dec/2012:00:01:43 +0100]
```

A questo punto, se il file non esiste, ci viene dato un chiaro messaggio di errore:

```
$ python argpar6.py foofile 2
usage: argpar6.py [-h] file index
argpar6.py: error: argument file: can't open 'foofile': [Errno 2] No such file or directory: 'foofile'
```

Argomenti opzionali e valori di default

Per default gli argomenti sono posizionali, nel senso che il primo viene

assegnato con la prima chiamata a `parser.add_argument()`, il secondo con la seconda chiamata, e via dicendo per i restanti. Non possiamo, quindi, invertire l'ordine degli argomenti:

```
$ python argpar6.py 2 django_pariglias_com.log
usage: argpar6.py [-h] file index
argpar6.py: error: argument file: can't open '2': [Errno 2] No such file or directory: '2'
```

Se volessimo specificare a quale attributo del namespace assegnare un argomento, dovremmo utilizzare le cosiddette *opzioni*, chiamate anche *switch*. Se, ad esempio, volessimo che un argomento preceduto dall'opzione `--file` venisse assegnato all'attributo `file` del namespace, e che uno preceduto dall'opzione `--index` venisse assegnato all'attributo `index`, non dovremmo far altro che chiamare `parser.add_argument()`, come mostrato di seguito:

```
$ cat argpar7.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--file', type=argparse.FileType(), help='Nome del file da aprire')
parser.add_argument('--index', type=int, help='Indice della linea del file da visualizzare')
args = parser.parse_args()
lines = args.file.readlines()
print(lines[args.index])
```

In questo modo l'ordine degli argomenti non è più posizionale, per cui, volendo, possiamo passare prima l'indice della linea e dopo il file:

```
$ python argpar7.py --index 2 --file django_pariglias_com.log
66.249.73.89 - [29/Dec/2012:00:01:43 +0100]
```

È possibile anche specificare una *short option*:

```
$ cat argpar8.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-f', '--file', type=argparse.FileType(), help='Nome del file da aprire')
parser.add_argument('-i', '--index', type=int, help='Indice della linea del file da visualizzare')
args = parser.parse_args()
lines = args.file.readlines()
print(lines[args.index])
```

```
$ python argpar8.py -i 2 -f django_pariglias_com.log
66.249.73.89 - [29/Dec/2012:00:01:43 +0100]
```

Per default gli argomenti associati alle opzioni non sono obbligatori, nel senso che non è necessario che vengano passati da linea di comando:

```
$ cat argpar9.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-f', '--file', type=argparse.FileType(),
help='Nome del file da aprire')
parser.add_argument('-i', '--index', type=int,
help='Indice della linea del file da visualizzare')
args = parser.parse_args()
print(args)
```

```
$ python argpar9.py
Namespace(file=None, index=None)
```

Nel caso dello script *argpar8.py*, però, la mancanza di un argomento dà luogo a un errore, oltretutto poco chiaro:

```
$ python argpar8.py
Traceback (most recent call last):
...
AttributeError: 'NoneType' object has no attribute 'readlines'
```

Sarebbe utile poter assegnare all'argomento omesso un valore di default, oppure poterlo dichiarare obbligatorio. Per assegnare un valore di default si utilizza il parametro `default` di `parser.add_argument()`:

```
$ cat argpar10.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-f', '--file', type=argparse.FileType(), help='Nome del file')
parser.add_argument('-i', '--index', type=int, default=0, help='Indice della linea')
args = parser.parse_args()
lines = args.file.readlines()
print(lines[args.index])
```

```
$ python argpar10.py -f django_pariglias_com.log
66.249.73.89 - [29/Dec/2012:00:01:06 +0100]
```

Per rendere obbligatorio un argomento associato a una opzione dobbiamo passare `required=True` al metodo `parser.add_argument()`:

```
$ cat argpar11.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-f', '--file', type=argparse.FileType(), required=True,
help='Nome del file da aprire')
parser.add_argument('-i', '--index', type=int, default=0, help='Indice della linea')
args = parser.parse_args()
lines = args.file.readlines()
```

```
print(lines[args.index])
```

In questo modo, se manca l'argomento obbligatorio, il parser blocca l'esecuzione del programma e mostra un utile messaggio di errore:

```
$ python argpar11.py
usage: argpar11.py [-h] -f FILE -i INDEX
argpar11.py: error: the following arguments are required: -f/--file, -i/--index
```

NOTA

Quando si parla di *argomenti opzionali* (*optional arguments*) non ci si riferisce agli argomenti che non sono obbligatori, ma a quelli associati a una opzione. Questi sono obbligatori se `required=True`, altrimenti per default possono anche essere omessi.

Valori all'interno di un certo range

Supponiamo ora di voler limitare la scelta della linea da visualizzare a una tra le prime cinque. Potremmo effettuare un test per verificare che si abbia `args.index` in `range(5)`, ma esiste una soluzione molto migliore, che consiste nell'assegnare il range di valori al parametro `choices`:

```
$ cat argpar12.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-f', '--file', type=argparse.FileType(), required=True,
help='Nome del file da aprire')
parser.add_argument('-i', '--index', type=int, default=0, choices=range(5),
help='Indice della linea')
args = parser.parse_args()
lines = args.file.readlines()
print(lines[args.index])
```

Questa soluzione presenta due vantaggi. Il primo è che il parser verifica che l'indice appartenga al range di valori consentiti, e, nel caso non lo sia, stampa un chiaro messaggio di errore:

```
$ python argpar12.py -f django_pariglias_com.log -i 10
usage: argpar12.py [-h] -f FILE [-i {0,1,2,3,4}]
argpar12.py: error: argument -i/--index: invalid choice: 10 (choose from 0, 1, 2, 3, 4)
```

Il secondo vantaggio è che il range di valori consentito viene mostrato anche nel messaggio di help:

```
$ python argpar12.py -h
usage: argpar12.py [-h] -f FILE [-i {0,1,2,3,4}]
```

optional arguments:

```
-h, --help show this help message and exit
-f FILE, --file FILE Nome del file da aprire
-i {0,1,2,3,4}, --index {0,1,2,3,4}
Indice della linea
```

A questo punto possiamo iniziare ad analizzare il file *webstats.py*.

Il parsing degli argomenti di webstats.py

Come abbiamo detto, le linee che si occupano del parsing degli argomenti sono le seguenti:

```
parser = argparse.ArgumentParser(
description="Processa un file avente i dati di accesso ad un sito web")
parser.add_argument('-f', '--file', type=argparse.FileType(), required=True,
help='File da processare')
parser.add_argument('-d', '--date', type=str2date, required=True,
help="Data: 'y m d'")
parser.add_argument('-r', '--range', default=0, type=int, choices=range(3))
args = parser.parse_args()
```

Le prime due linee dovrebbero essere chiare, per cui passiamo direttamente alla terza. In questa viene passato al parametro `type` la funzione `str2date` definita qualche riga sopra, che qui riportiamo:

```
>>> from datetime import date
>>> def str2date(s):
...     return date(*[int(item) for item in s.split()])
```

Questa funzione prende una stringa 'anno mese giorno' e restituisce un oggetto di tipo `datetime.date`:

```
>>> str2date('2013 11 28')
datetime.date(2013, 11, 28)
```

Usare un oggetto `datetime.date` piuttosto che una stringa ci consente di effettuare delle utili operazioni e ci permette, inoltre, di verificare la validità della data:

```
>>> d = datetime.date(2013, 2, 28)
>>> type(d)
<class 'datetime.date'>
>>> d = datetime.date(2013, 2, 29) # Il 2013 non è un anno bisestile...
```

```
Traceback (most recent call last):
...
ValueError: day is out of range for month
```

Come possiamo vedere, la classe `datetime.date` non prende una stringa come argomento, ma tre interi rappresentativi, rispettivamente, dell'anno, del mese e del giorno. Vediamo di capire come abbiamo ottenuto i tre interi a partire dalla stringa 'anno mese giorno'.

Il primo passo da compiere è spezzare la stringa in modo da ottenere i tre elementi, per poi convertire questi ultimi in interi. Per fare ciò abbiamo utilizzato una list comprehension:

```
>>> [int(item) for item in '2012 12 31'.split()]
[2012, 12, 31]
```

Come abbiamo detto, la classe `datetime.date` prende tre interi, per cui non le possiamo passare direttamente la lista come argomento:

```
>>> datetime.date([int(item) for item in '2012 12 31'.split()])
Traceback (most recent call last):
...
TypeError: an integer is required
```

La soluzione consiste nel far precedere la lista da un asterisco. Come vedremo nel [Capitolo 3](#), quando parleremo delle funzioni, l'asterisco davanti all'argomento indica che questo deve essere spaccettato, e ciascun elemento viene quindi passato alla funzione come argomento singolo:

```
>>> def foo(a, b, c):
...     print(a, b, c)
...
>>> foo(*[1, 2, 3]) # Equivalente a 'foo(1, 2, 3)'
1 2 3
```

Ovviamente questo è possibile solo se l'oggetto è iterabile. La chiamata:

```
>>> datetime.date(*[int(item) for item in '2012 12 31'.split()])
datetime.date(2012, 12, 31)
```

è equivalente alla seguente:

```
>>> datetime.date(*[2012, 12, 31])
datetime.date(2012, 12, 31)
```

E, in base a quanto abbiamo appena detto, è equivalente alla seguente:

```
>>> datetime.date(2012, 12, 31)
datetime.date(2012, 12, 31)
```

Le ultime due linee sono le seguenti:

```
parser.add_argument('-r', '--range', default=0, type=int, choices=range(3))
args = parser.parse_args()
```

Su queste non c'è nulla da aggiungere, visto che abbiamo già detto tutto nelle precedenti sezioni. Per approfondire la conoscenza del modulo `argparse` si può consultare la documentazione online della libreria standard all'indirizzo <http://docs.python.org/3/library/argparse.html>, oppure il tutorial presente sul sito ufficiale, alla pagina <http://docs.python.org/3/howto/argparse.html>.

Lavorare con le date

Vediamo adesso di analizzare le seguenti linee di codice:

```
total_hits = []

format_ = '%d/%b/%Y:%H:%M:%S %z' # day/month/year:hour:minute:second time-zone
for line in args.file:
    ip, hit_time = line.split(' - ')
    hit_date = datetime.strptime(hit_time.strip(), format_)
    if abs(args.date - hit_date.date()) <= timedelta(days=args.range):
        total_hits.append(ip) # Aggiungi l'indirizzo IP alla lista
```

La lista assegnata all'etichetta `total_hits` serve per memorizzare gli indirizzi IP di tutti gli hit relativi al periodo indicato. Passiamo subito al ciclo `for`. L'istruzione `for` itera sulle linee del file e per ciascuna di esse assegna l'indirizzo IP all'etichetta `ip` e la data all'etichetta `hit_time`, come mostra il seguente esempio:

```
>>> line = next(open('django_parigias_com.log')) # Assegna a 'line' la prima linea
>>> line
'66.249.73.89 - [29/Dec/2012:00:01:06 +0100]\n'
>>> ip, hit_time = line.split(' - ')
>>> ip # Stringa che rappresenta l'indirizzo IP dell'hit
'66.249.73.89'
>>> hit_time # Stringa che rappresenta la data di accesso
'[29/Dec/2012:00:01:06 +0100]\n'
```

Veniamo adesso alla linea che utilizza l'etichetta `format_`:

```
hit_date = datetime.strptime(hit_time.strip(), format_)
```

La funzione `datetime.datetime.strptime()` prende una stringa contenente un formato e restituisce un oggetto di tipo `datetime.datetime`. Vediamo qualche esempio:

```
>>> datetime.strptime('31-12-2012', '%d-%m-%Y') # 'day-month-year'
datetime.datetime(2012, 12, 31, 0, 0)
>>> datetime.strptime('31-12-2012:11:31:59', '%d-%m-%Y:%H:%M:%S')
datetime.datetime(2012, 12, 31, 11, 31, 59)
```

Nel nostro script abbiamo passato a `datetime.strptime()` la data priva del carattere di `newline`:

```
>>> hit_time
'[29/Dec/2012:00:01:06 +0100]\n'
>>> hit_time.strip() # Rimuoviamo il carattere di newline
'[29/Dec/2012:00:01:06 +0100]'
```

Nelle date contenute nei file di log il mese non è indicato in formato numerico, ma mediante un'abbreviazione del nome. In questo caso la direttiva da utilizzare non è `%m` ma `%b`. Nella data, inoltre, è presente anche il *time-zone*, e questo va specificato con la direttiva `%z`. Quindi, in definitiva:

```
>>> format_ = '%d/%b/%Y:%H:%M:%S %z'
>>> hit_date = datetime.strptime(hit_time.strip(), format_)
>>> hit_date
datetime.datetime(2012, 12, 29, 0, 1, 6, tzinfo=datetime.timezone(datetime.timedelta(0, 3600)))
```

NOTA

Quando definiamo un'etichetta dovremmo prestare attenzione a non sovrascriverne una built-in, altrimenti è facile poi commettere qualche errore. Se, ad esempio, volessimo utilizzare una stringa che rappresenta un formato, l'etichetta più adatta sarebbe `format`, ma, poiché questa è già definita e assegnata alla funzione built-in `format()`, sarebbe meglio se le dessimo un nome diverso. La prassi più comune è quella di adottare, anche in questo caso, la regola suggerita dalla PEP-0008 per i conflitti tra etichette e parole chiave, ovvero aggiungere un underscore: `format_`.

Vediamo ora il significato dell'espressione `abs(args.date - hit_date.date()) <= timedelta(days=args.range)`. Questa calcola la differenza in valore assoluto tra due date e restituisce `True` se questa differenza è inferiore o uguale al range di giorni

passato al programma con `--range`. Vediamo qualche esempio:

```
>>> date(2012, 12, 15) - date(2012, 12, 13) <= timedelta(2)
True
>>> date(2012, 12, 15) - date(2012, 12, 13) <= timedelta(1)
False
```

Se, però, la seconda data è superiore alla prima, la differenza sarà negativa e quindi la disuguaglianza risulterà sempre verificata:

```
>>> date(2012, 12, 15) - date(2012, 12, 17)
datetime.timedelta(-2)
>>> date(2012, 12, 15) - date(2012, 12, 17) <= timedelta(1)
True
```

Per fare in modo che la disuguaglianza sia verificata solo quando la seconda data si riferisce ai giorni che vanno dal 13 dicembre 2012 al 17 dicembre 2012, dobbiamo considerare la differenza delle date in valore assoluto:

```
>>> abs(date(2012, 12, 15) - date(2012, 12, 17))
datetime.timedelta(2)
>>> abs(date(2012, 12, 15) - date(2012, 12, 17)) <= timedelta(1)
False
>>> abs(date(2012, 12, 15) - date(2012, 12, 17)) <= timedelta(2)
True
>>> abs(date(2012, 12, 15) - date(2012, 12, 13)) <= timedelta(2)
True
```

Visto che ci interessa una differenza in termini di giorni, possiamo assegnare esplicitamente il numero di giorni al parametro `days` di `timedelta`, in modo che il codice risulti ancora più chiaro:

```
>>> abs(date(2012, 12, 15) - date(2012, 12, 13)) <= timedelta(days=2)
True
>>> abs(date(2012, 12, 15) - date(2012, 12, 13)) <= timedelta(days=1)
False
```

NOTA

La classe `timedelta` consente di comparare non solo differenze di oggetti `datetime.date`, ma anche di oggetti `datetime.timedelta`. Per questo motivo accetta come argomento, oltre al numero di giorni, anche il numero di secondi (`seconds`) o microsecondi (`microseconds`):

```
>>> from datetime import datetime as dt
```

```
>>> from datetime import timedelta
>>> dt(2012, 12, 15, 11, 31, 59) - dt(2012, 12, 15, 11, 31, 50)
datetime.timedelta(0, 9)
>>> dt(2012, 12, 15, 11, 31, 59) - dt(2012, 12, 15, 11, 31, 50) \
... <= timedelta(seconds=9)
True
>>> dt(2012, 12, 15, 11, 31, 59) - dt(2012, 12, 15, 11, 31, 50) \
... <= timedelta(seconds=8)
False
```

Tornando allo script *webstats.py*, `hit_date` non è un oggetto di tipo `datetime.date`, ma un oggetto di tipo `datetime.datetime`, contenente, oltre alle informazioni su giorno, mese e anno, anche quelle sull'ora, i minuti, i secondi e il time-zone.

Poiché `args.date` è un oggetto di tipo `datetime.date`, per poter effettuare la differenza tra `args.date` e `hit_date` dobbiamo ottenere da quest'ultimo un oggetto `datetime.date`. Il metodo `datetime.datetime.date()` fa proprio questo, per cui `hit_date.date()` restituisce un oggetto di tipo `datetime.date` che ha lo stesso anno, mese e giorno di `hit_date`:

```
>>> hit_date.date()
datetime.date(2012, 12, 29)
```

A questo punto dovrebbe esserci perfettamente chiaro ciò che fa il seguente codice:

```
if abs(args.date - hit_date.date()) <= timedelta(days=args.range):
    total_hits.append(ip) # Aggiungi l'indirizzo IP alla lista
```

La lista `total_hits` contiene tutti gli indirizzi IP relativi agli hit del periodo selezionato. Un indirizzo IP può comparire più volte nella lista, sia perché la richiesta di una pagina da parte di un visitatore può dare luogo a diversi hit, sia perché un visitatore può richiedere più di una pagina. Per ottenere il numero di visitatori unici dobbiamo considerare ciascun indirizzo IP una sola volta, e questo è esattamente ciò che fa il set:

```
>>> total_hits
['66.249.73.89', '66.249.73.89', '69.11.29.112', '65.55.21.198', '65.55.21.198']
>>> set(total_hits)
{'65.55.21.198', '66.249.73.89', '69.11.29.112'}
```

Anche il significato della linea di codice `unique_visitors = set(total_hits)` è adesso chiaro.

Il modulo `collections` della libreria standard

Come possiamo vedere, nelle seguenti linee di codice facciamo uso della

classe `collections.Counter`:

```
counter = collections.Counter()
for ip_address in total_hits:
    counter[ip_address] += 1
```

Il modulo `collections` fornisce il supporto per lavorare con le collezioni di oggetti. Se ricordiamo, in questo capitolo abbiamo già visto qualche esempio di utilizzo di questo modulo (tuple con nome e dizionari ordinati).

Il nostro scopo è quello di avere un oggetto *dictionary-like* che abbia per chiavi gli indirizzi IP e per valori i corrispondenti hit. Ad esempio, supponiamo di avere la seguente lista di indirizzi IP:

```
>>> total_hits
['66.249.73.89', '66.249.73.89', '69.11.29.112', '65.55.21.198', '65.55.21.198']
```

Vorremmo creare un dizionario che contenga per chiavi i tre indirizzi unici '65.55.21.198', '66.249.73.89' e '69.11.29.112' e come valori i corrispondenti hit. Una soluzione potrebbe essere la seguente:

```
>>> counter = {}
>>> unique_visitors = set(total_hits)
>>> for ip in unique_visitors:
...     counter[ip] = total_hits.count(ip)
...
>>> counter
{'65.55.21.198': 2, '66.249.73.89': 2, '69.11.29.112': 1}
```

Ora ci interessa mostrare i primi due visitatori con il maggior numero di hit. Una soluzione potrebbe essere quella di ordinare le coppie chiave valore con la funzione built-in `sorted()`, utilizzando come chiave una funzione che prende una sequenza e ne restituisce l'ultimo elemento:

```
>>> def last(seq):
...     return seq[-1] # Restituisci l'ultimo elemento della sequenza
...
>>> sorted_counter = sorted(counter.items(), key=last, reverse=True)
>>> sorted_counter
[('65.55.21.198', 2), ('66.249.73.89', 2), ('69.11.29.112', 1)]
```

A questo punto basta prendere `sorted_counter[0:2]`. La classe `collection.Counter` consente di eseguire tutto questo lavoro in modo più immediato, più conciso e più semplice da capire:

```
>>> counter = collections.Counter() # Creiamo un particolare dizionario vuoto
>>> for ip_address in total_hits:
...     counter[ip_address] += 1
...
>>> counter
Counter({'65.55.21.198': 2, '66.249.73.89': 2, '69.11.29.112': 1})
```

Per ottenere i due indirizzi IP con maggior numero di hit chiamiamo `counter.most_common(2)`:

```
>>> counter.most_common(2) # Restituisce i due elementi più comuni
[('65.55.21.198', 2), ('66.249.73.89', 2)]
```

Adesso possiamo rileggere lo script *webstats.py*. Ci renderemo conto che tutto ciò che inizialmente ci sembrava incomprensibile adesso ha un senso.

Funzioni, generatori e file

Nel capitolo precedente abbiamo approfondito lo studio del core data type di Python e introdotto alcuni importanti moduli della libreria standard. In questo capitolo continueremo sulla stessa linea: approfondiremo lo studio delle **funzioni** e dei **file** e vedremo alcuni importanti **moduli della libreria standard** che ci consentiranno di utilizzare le **wildcard** e le **espressioni regolari**.

Definizione e chiamata di una funzione

Come ormai sappiamo, le funzioni vengono create con la parola chiave `def`, seguita dall'etichetta e dalle parentesi tonde. Se la funzione definisce dei parametri, questi vanno elencati tra le parentesi, separandoli con una virgola:

```
>>> def average(obj):
...     """Restituisci il valore medio tra gli elementi di `obj`"""
...     return sum(obj) / len(obj)
...
>>> average([1, 7, 3, 5])
4.0
```

Come abbiamo già visto nella sezione *La funzione built-in `help()` e le stringhe di documentazione* del [Capitolo 1](#), quando la prima istruzione di una funzione è un letterale stringa, questa viene detta *docstring* e si può accedere a essa mediante l'attributo `__doc__` della funzione:

```
>>> average.__doc__
'Restituisci il valore medio degli elementi di `obj`'
```

Le funzioni sono degli oggetti di tipo `types.FunctionType`:

```
>>> type(average)
<class 'function'>
>>> import types
>>> types.FunctionType
<class 'function'>
```

e la loro etichetta è un riferimento all'oggetto funzione:

```
>>> av = average
>>> average = 33
>>> average # L'etichetta `average` non fa più riferimento alla funzione
33
>>> av # L'etichetta `av` fa riferimento alla funzione
<function average at 0xb7334344>
>>> av.__name__
'average'
>>> av([2, 2, 2, 2])
2.0
```

Questo può essere verificato anche guardando il bytecode:

```
>>> import dis
>>> dis.dis('def foo(): pass')
1 0 LOAD_CONST 0 (<code object foo at 0xb7391660, file "<dis>", line 1>)
3 LOAD_CONST 1 ('foo')
6 MAKE_FUNCTION 0
9 STORE_NAME 0 (foo)
12 LOAD_CONST 2 (None)
15 RETURN_VALUE
```

Da questo deduciamo che è stato creato un oggetto funzione, al quale è stato dato il nome 'foo'. Questo oggetto è poi stato assegnato all'etichetta `foo` con l'istruzione `STORE_NAME`.

La `def` può comparire in ogni punto del programma in cui è ammessa un'istruzione composta, quindi anche all'interno di altre istruzioni:

```
>>> if True:
...     def foo():
...         print('I am foo()')
...
>>> foo()
I am foo()
```

Una funzione restituisce sempre un oggetto (il *risultato*), in modo *implicito* o *esplicito*. Per restituire un oggetto in modo esplicito, si utilizza l'istruzione `return`:

```
>>> def foo():
...     return 100
...
>>> foo()
100
>>> import dis
>>> dis.dis(foo) # La prima istruzione carica la costante `100`, la seconda la restituisce
2 0 LOAD_CONST 1 (100)
3 RETURN_VALUE
```

Se, durante il flusso di esecuzione della funzione, non viene eseguita l'istruzione `return`, allora la funzione termina la sua esecuzione restituendo *implicitamente* l'oggetto `None`:

```
>>> def foo():
```

```

... pass
...
>>> print(foo()) # Viene restituito implicitamente `None`
None
>>> import dis
>>> dis.dis(foo) # La prima istruzione carica None, la seconda lo restituisce
2 0 LOAD_CONST 0 (None)
3 RETURN_VALUE

```

L'istruzione `return` restituisce un singolo oggetto, per cui, se si ha la necessità di restituirne più di uno, si può utilizzare la strategia di impacchettarli in una tupla o in una lista e restituire quest'ultima:

```

>>> def foo():
...     a = 33
...     b = 'python'
...     c = (1, 2, 3)
...     return a, b, c # Impacchettiamo i tre oggetti in una tupla
...
>>> foo() # Restituisce una tupla
(33, 'python', (1, 2, 3))
>>> a, b, c = foo()
>>> a
33
>>> b
'python'
>>> c
(1, 2, 3)

```

Parametri, argomenti e valori di ritorno

Le etichette dichiarate tra le parentesi tonde dell'istruzione `def` vengono chiamate *parametri*, mentre gli oggetti assegnati ai parametri al momento della chiamata alla funzione vengono detti *argomenti*:

```

>>> def foo(a, b): # Le etichette `a` e `b` sono i parametri della funzione
...     print(a, b)
...
>>> foo(1, 2) # I letterali `1` e `2` sono gli argomenti della funzione
(1, 2)
>>> foo(['python', 3], sum) # La lista e `sum` sono gli argomenti
(['python', 3], <built-in function sum>)

```

L'insieme dei parametri dichiarati nella definizione della funzione è chiamata *signature*. Ad esempio, la signature della precedente funzione `foo()` è `(a, b)`. Approfondiremo questo discorso più avanti, quando parleremo del modulo

inspect della libreria standard.

Python ha una sola modalità di assegnamento degli argomenti ai parametri: quella per *riferimento*. Consideriamo, ad esempio, la seguente funzione `foo()`:

```
>>> def foo(a, b):
...     print(a, id(a), sep=' -> ')
...     print(b, id(b), sep=' -> ')
```

Se le passiamo due oggetti, le etichette `a` e `b` locali alla funzione faranno riferimento a essi:

```
>>> num = 55
>>> mylist = [1, 2, 3]
>>> id(num), id(mylist)
(8737728, 140077235270072)
>>> foo(num, mylist)
55 -> 8737728
[1, 2, 3] -> 140077235270072
```

In pratica, gli argomenti passati alla funzione vengono implicitamente assegnati ai parametri, per cui le etichette `a` e `num` fanno riferimento al medesimo oggetto, avente identità `8737728`, e le etichette `b` e `mylist` fanno entrambe riferimento all'oggetto con identità `140077235270072`.

Ecco un altro esempio:

```
>>> def foo(a, b):
...     a = 'python' # Assegno la stringa 'python' all'etichetta locale `a`
...     b.append(100) # Modifico l'oggetto a cui `b` fa riferimento
...
>>> obj1 = 33
>>> obj2 = [1, 2, 3]
>>> foo(obj1, obj2)
>>> obj1 # `obj1` fa riferimento al numero intero 33 e non alla stringa 'python'
33
>>> obj2 # La lista è stata modificata con l'aggiunta del numero 100
[1, 2, 3, 100]
>>> foo(obj1, obj2)
>>> obj2
[1, 2, 3, 100, 100]
```

Se non vogliamo passare alla funzione un riferimento all'oggetto originario, allora dobbiamo creare una copia dell'oggetto e passare quello alla funzione. Ad esempio, se l'oggetto è una lista, possiamo usare il metodo `list.copy()`:

```
>>> foo(obj1, obj2.copy()) # Passiamo una copia della lista
```

```
>>> obj2 # La funzione non ha modificato la lista
[1, 2, 3, 100, 100]
```

oppure lo slicing:

```
>>> foo(obj1, obj2[:]) # Ancora una volta passiamo una copia della lista
>>> obj2 # La funzione non ha modificato la lista
[1, 2, 3, 100, 100]
```

Anche il passaggio del valore di ritorno avviene sempre per riferimento:

```
>>> def foo(a):
...     a.append(100)
...     return a
...
>>> mylist1 = [1, 2, 3]
>>> mylist2 = foo(mylist1)
>>> mylist1
[1, 2, 3, 100]
>>> mylist2
[1, 2, 3, 100]
```

Come ormai sappiamo, una funzione può prendere argomenti di qualsiasi tipo e restituire oggetti di qualsiasi tipo:

```
>>> def foo(a):
...     return a
...
>>> foo('python') # Passo una stringa
'python'
>>> foo([1, 2, 3]) # Passo una lista
[1, 2, 3]
>>> foo(sum) # Passo una funzione
<built-in function sum>
```

Modalità di accoppiamento degli argomenti ai parametri

La modalità predefinita per l'accoppiamento (*matching*) degli argomenti ai parametri è per posizione, da sinistra a destra:

```
>>> def foo(par1, par2, par3):
...     print('par1: ', par1)
...     print('par2: ', par2)
...     print('par3: ', par3)
...
>>> foo('python', 2, sum)
par1: python
```

```
par2: 2
par3: <built-in function sum>
```

Nel matching per posizione il numero degli argomenti deve essere pari a quello dei parametri:

```
>>> foo(1, 2, 3, 4) # Numero degli argomenti superiore a quello dei parametri
Traceback (most recent call last):
...
TypeError: foo() takes 3 positional arguments but 4 were given
>>> foo(1, 2) # Numero degli argomenti inferiore a quello dei parametri
Traceback (most recent call last):
...
TypeError: foo() missing 1 required positional argument: 'par3'
```

Vi sono modalità di matching degli argomenti ai parametri alternative a quella predefinita. Queste consentono di effettuare l'accoppiamento in diversi modi, che tratteremo di seguito. L'argomento è forse un po' noioso e molti dettagli sono difficili da memorizzare, per cui probabilmente sarete tentati di saltare qualche sezione. Dopo aver studiato le prime due sezioni, molto importanti, che riguardano il matching per nome e il matching con valori di default, potete prendervi la licenza di farlo. Bisogna comunque tenere presente che comprendere come avviene il matching degli argomenti ai parametri è importante, per cui prima o poi sarete costretti a tornare sui vostri passi e riprendere anche le parti più noiose. Al termine delle prossime due sezioni, prima di decidere se approfondire anche le restanti riguardanti il matching degli argomenti, ci concederemo una piccola e divertente pausa, riguardante un *easter egg*.

Matching per nome

Questa modalità, detta *matching per nome* (*matching by keyword*), consente di specificare il parametro al quale assegnare l'argomento:

```
>>> def foo(a, b, c, d):
...     print(a, b, c, d, sep=' | ')
...
>>> foo(b=2, a=1, d=4, c=3)
1 | 2 | 3 | 4
```

Come possiamo osservare da questo esempio, è sufficiente assegnare in modo esplicito l'argomento al parametro al momento della chiamata a funzione. Le modalità di matching per posizione e per nome possono essere utilizzate

insieme:

```
>>> foo(1, 2, d=4, c=3)
1 | 2 | 3 | 4
>>> foo(1, c=3, d=4, b=2)
1 | 2 | 3 | 4
```

Si osservi, però, come gli assegnamenti per posizione debbano sempre precedere quelli per nome:

```
>>> foo(1, 2, c=3, 4)
...
SyntaxError: non-keyword arg after keyword arg
```

Matching con valori di default

Come possiamo immaginare, è possibile specificare dei valori di default. Questi vengono assegnati ai parametri in modo automatico qualora non gli vengano passati esplicitamente degli argomenti al momento della chiamata alla funzione:

```
>>> def foo(a=1, b=2, c=3, d=4):
...     print(a, b, c, d, sep=' | ')
...
>>> foo()
1 | 2 | 3 | 4
```

Le modalità di matching per posizione, per nome e con valori di default possono essere combinate assieme:

```
>>> foo(c=33, a=11, d=44) # `a`, `c` e `d` assegnati per nome, `b` per default
11 | 2 | 33 | 44
>>> foo(11) # Il parametro `a` è assegnato per posizione, gli altri per default
11 | 2 | 3 | 4
>>> foo(11, 22, d=44) # Primi due per posizione, c per default, d per nome
11 | 22 | 3 | 44
>>> def foo(a, b, c=3, d=4):
...     print(a, b, c, d, sep=' | ')
...
>>> foo(11, d='quarto', b='secondo') # `b` e `d` per nome, `c` per default
11 | secondo | 3 | quarto
```

Nella definizione di una funzione, i parametri con valori di default non possono precedere quelli senza:

```
>>> def foo(a=1, b, c, d):
```

```
... print(a, b, c, d, sep=' | '))
...
File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Quando un parametro ha un valore di default, ogniqualvolta viene chiamata la funzione senza che le venga specificato il corrispondente argomento, al parametro viene assegnato sempre il medesimo oggetto e non uno nuovo. Possiamo disinteressarci di questo comportamento quando il valore di default è un oggetto immutabile, ma dobbiamo invece prestare attenzione nel caso di un oggetto mutabile:

```
>>> def foo(a=[1, 2, 3]):
...     a.append(100)
...     print(a)
...
>>> foo()
[1, 2, 3, 100]
>>> foo()
[1, 2, 3, 100, 100]
>>> foo([44])
[44, 100]
>>> foo()
[1, 2, 3, 100, 100, 100]
>>> foo()
[1, 2, 3, 100, 100, 100, 100]
```

Ecco un altro esempio:

```
>>> mylist = [1, 2, 3]
>>> def foo(a=mylist):
...     print(a)
...
>>> foo(44)
44
>>> foo()
[1, 2, 3]
>>> mylist.append(4)
>>> foo()
[1, 2, 3, 4]
```

Tipicamente questo comportamento riserva una sorpresa ai programmatori Python alle prime armi, poiché si tende a pensare al valore di default come a un inizializzatore. Nel caso seguente, ad esempio:

```
>>> def foo(arg=[], value=0):
...     arg.append(value)
```

```
... return arg
```

solitamente ci si aspetta che `foo()` restituisca la lista `[0]` quando viene chiamata senza argomenti:

```
>>> foo()
[0]
```

Però, quando si chiama `foo()` una seconda volta, solitamente si resta sorpresi:

```
>>> foo()
[0, 0]
>>> foo()
[0, 0, 0]
```

Una soluzione può essere la seguente:

```
>>> def foo(arg=None, value=0):
...     arg = [] if arg == None else arg
...     arg.append(value)
...     return arg
...
>>> foo()
[0]
>>> foo()
[0]
>>> foo([1])
[1, 0]
>>> foo()
[0]
```

Come promesso, prima di andare avanti concediamoci una piccola e divertente pausa. Importiamo il modulo `antigravity`:

```
>>> import antigravity
```

Alla pagina <http://python-history.blogspot.it/2010/06/import-antigravity.html> Guido racconta la storia di questo easter egg.

Matching con varargs

Quando, nella definizione di una funzione, un parametro viene fatto precedere da un asterisco, allora tutti gli argomenti passati alla funzione vengono impacchettati in una tupla che viene poi assegnata al parametro:

```
>>> def foo(*varargs):
...     print(varargs)
...
>>> foo() # Viene assegnata a `varargs` una tupla vuota
()
>>> foo('a')
('a',)
>>> foo(1, 2, 3, 4)
(1, 2, 3, 4)
>>> foo(['a', 'b'], 1, 2, 3, 'python')
(['a', 'b'], 1, 2, 3, 'python')
```

Le funzioni di questo tipo, che possono prendere un numero arbitrario di argomenti, vengono dette *funzioni variadic* (*variadic functions*).

NOTA

Il parametro al quale viene assegnata la tupla è abitualmente chiamato *varargs*, nome che deriva da quello del vecchio file header C, *varargs.h*, che forniva gli strumenti per scrivere liste variabili di argomenti. Talvolta lo si indica in modo più conciso con *args*.

Questa modalità può essere combinata con quella posizionale e con gli assegnamenti di default:

```
>>> def foo(a, b=2, *varargs):
...     print(a, b, varargs, sep=' | ')
...
>>> foo(1)
1 | 2 | ()
>>> foo(1, 'due')
1 | due | ()
>>> foo(1, 'due', 11, 22, 33)
1 | due | (11, 22, 33)
>>> foo(1, 2, ['python', 'py'], 222, sum)
1 | 2 | (['python', 'py'], 222, <built-in function sum>)
```

Si osservi come il matching per posizione preceda quello dei `varargs`:

```
foo(3, 4, 5, a=1, b=2)
Traceback (most recent call last):
...
TypeError: foo() got multiple values for argument 'a'
```

Infatti in questo caso è stato assegnato `a=3`, poi `b=4` e `varargs=(5,)`, e poi nuovamente

$a=1$, per cui si ha un assegnamento multiplo ad a .

Matching keyword-only

Se nella definizione della funzione i `varargs` precedono altri parametri, allora questi ultimi durante la chiamata devono essere passati dopo i `vararg` e solamente con matching per keyword:

```
>>> def foo(*varargs, a, b):
...     print(varargs)
...     print(a, b)
...
>>> foo(3, 4, 5, a=1, b=2)
(3, 4, 5)
1 2
>>> foo(b=2, a=1)
()
1 2
>>> foo(a=1, b=2, 3, 4, 5) # Errore: dobbiamo assegnare i parametri `a` e `b` dopo i varargs
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Poiché i parametri che seguono `varargs` possono essere assegnati solo per nome, questa modalità di matching è detta *keyword-only*. Ecco un altro esempio:

```
>>> def foo(a, *varargs, b, c):
...     print(a, varargs, b, c, sep=' | ')
...
>>> foo(1, 11, 22, 33, c=3, b=2)
1 | (11, 22, 33) | 2 | 3
>>> foo(1, 2, 3) # Errore: dobbiamo assegnare i parametri `b` e `c` per nome
Traceback (most recent call last):
...
TypeError: foo() missing 2 required keyword-only arguments: 'b' and 'c'
```

Il matching keyword-only può avvenire anche utilizzando dei valori di default:

```
>>> def foo(a, *varargs, b=2, c=3):
...     print(a, varargs, b, c, sep=' | ')
...
>>> foo(1)
1 | () | 2 | 3
>>> foo(1, 11, 22, 33, 44)
1 | (11, 22, 33, 44) | 2 | 3
>>> foo(1, 11, 22, c='tre', b='due')
1 | (11, 22) | due | tre
```


Il matching keyword-only può essere utilizzato anche in assenza di un parametro `varargs`. Per fare ciò si definisce un parametro rappresentato da un solo asterisco, stando a indicare che tutti i parametri che seguono devono essere assegnati in modalità keyword-only:

```
>>> def foo(*, a, b, c):
...     print(a, b, c)
...
>>> foo(1, 2, 3)
Traceback (most recent call last):
...
TypeError: foo() takes 0 positional arguments but 3 were given
>>> foo(b=2, c=3, a=1)
1 2 3
```

Anche questa modalità può essere utilizzata in combinazione con le altre:

```
>>> def foo(a, b=2, *, c, d=4):
...     print(a, b, c, d)
...
>>> foo(1, c=3)
1 2 3 4
>>> foo(1, 'due', d='quattro', c=3)
1 due 3 quattro
>>> foo(1, 'due', 3, 4)
Traceback (most recent call last):
...
TypeError: foo() takes from 1 to 2 positional arguments but 4 were given
```

Il matching keyword-only è disponibile a partire da Python 3 ed è discusso nella PEP-3102.

Matching con spaccettamento di oggetti iterabili

Abbiamo già visto questa modalità nell'esercizio conclusivo al termine del Capitolo 2. Come sappiamo, essa ci consente di *spacchettare* un oggetto iterabile, in modo che ogni suo elemento diventi un argomento da assegnare in modo posizionale al corrispondente parametro. Per fare ciò l'oggetto iterabile viene fatto precedere da un asterisco al momento della chiamata alla funzione:

```
>>> def foo(a, b, c):
...     print(a, b, c)
...
>>> foo(*[11, 22, 33])
11 22 33
```

Il numero degli elementi dell'oggetto iterabile deve essere pari a quello dei parametri della funzione:

```
>>> foo(*(1, 2, 3, 4))
Traceback (most recent call last):
...
TypeError: foo() takes 3 positional arguments but 4 were given
>>> foo(*(1, 2))
Traceback (most recent call last):
...
TypeError: foo() missing 1 required positional argument: 'c'
```

Lo spaccettamento è estremamente potente quando applicato a un parametro `varargs`, poiché possiamo disinteressarci sia del numero dei parametri sia di quello degli argomenti:

```
>>> def foo(*varargs):
...     for arg in varargs:
...         print(arg, end=' ')
...
>>> foo(*range(5))
0 1 2 3 4
```

Matching con kwargs

Quando, nella definizione di una funzione, un parametro `kwargs` viene fatto precedere da due asterischi, gli argomenti passati per nome nelle posizioni che competono a quel parametro vengono impacchettati in un dizionario e questo viene assegnato al parametro `kwargs`. Il dizionario avrà come chiavi i nomi dei parametri, e come valori i corrispondenti argomenti:

```
>>> def foo(**kwargs):
...     print(kwargs)
...
>>> foo()
{}
>>> foo(a=11, c=33, d=44, b=22)
{'d': 44, 'a': 11, 'b': 22, 'c': 33}
```

Questa tipologia di matching è detta *kwargs*, poiché solitamente questo è il nome che viene dato al parametro. Possiamo utilizzare la modalità `kwargs` combinata con le altre:

```
>>> def foo(a, b='secondo', *varargs, **kwargs):
...     print(a, b, varargs, kwargs, sep=' | ')
```

```
...
>>> foo(1)
1 | secondo | () | {}
>>> foo(1, 2)
1 | 2 | () | {}
>>> foo(b=22, a=11)
11 | 22 | () | {}
>>> foo(1, 2, 111, 222, c=11, d=22)
1 | 2 | (111, 222) | {'c': 11, 'd': 22}
```

Si tenga presente, però, che questo parametro va sempre posto in coda a tutti gli altri:

```
>>> def foo(a, b='secondo', **kwargs, *vargs):
    File "<stdin>", line 1
def foo(a, b='secondo', **kwargs, *vargs):
^
SyntaxError: invalid syntax
>>> def foo(a, **kwargs, b=33):
    File "<stdin>", line 1
def foo(a, **kwargs, b=33):
^
SyntaxError: invalid syntax
```

I parametri con matching keyword-only vanno quindi definiti prima di `kwargs`:

```
>>> def foo(a, *, b, c, **kwargs):
...     print(a, b, c, kwargs)
...
>>> foo(1, c=3, b=2)
1 2 3 {}
>>> foo(1, c=3, b=2, d=4, e=5, nome='python')
1 2 3 {'e': 5, 'd': 4, 'nome': 'python'}
>>> foo(1, 2, 3)
Traceback (most recent call last):
...
TypeError: foo() takes 1 positional argument but 3 were given
```

Questa modalità ci consente di assegnare dei valori di default ai parametri, in un modo alternativo a quello che già conosciamo. Consideriamo l'esempio seguente, nel quale il parametro keyword-only `b` ha come valore di default `3`:

```
>>> def foo(a, *, b='3'):
...     print(a, b)
...
>>> foo('Python', b='2')
Python 2
>>> foo('Python')
Python 3
>>> foo('Python', '2') # b deve essere assegnato con keyword-only
```

Traceback (most recent call last):

...

TypeError: foo() takes 1 positional argument but 2 were given

Possiamo ottenere lo stesso risultato utilizzando la seguente strategia:

```
>>> def foo(a, **kwargs):
...     b = kwargs.get('b', '3') # Se la chiave 'b' non esiste viene restituito '3'
...     print(a, b)
...
>>> foo('Python', b='2')
Python 2
>>> foo('Python')
Python 3
>>> foo('Python', '2')
Traceback (most recent call last):
...
TypeError: foo() takes 1 positional argument but 2 were given
```

Matching con spaccettamento di oggetti dictionary-like

Con questa modalità è possibile *spacchettare* un oggetto dictionary-like, in modo che ogni sua chiave indichi un parametro e il corrispondente valore diventi l'argomento. Per fare ciò l'oggetto dictionary-like deve essere preceduto da una coppia di asterischi al momento della chiamata alla funzione:

```
>>> def foo(a, b, c):
...     print(a, b, c)
...
>>> foo(**{'b': 22, 'a': 11, 'c': 33})
11 22 33
>>> import shelve
>>> myshelf = shelve.open('myfile')
>>> myshelf['a'] = 11
>>> myshelf['b'] = 22
>>> myshelf['c'] = 33
>>> foo(**myshef)
11 22 33
```

A ogni chiave deve corrispondere il nome di un parametro:

```
>>> foo(**{'b': 22, 'a': 11, 'c': 33, 'd': 44})
Traceback (most recent call last):
...
TypeError: foo() got an unexpected keyword argument 'd'
```

Lo spaccettamento consente anche l'assegnamento dei parametri keyword-

only:

```
>>> def foo(*, nome, nickname, anno):  
...     print(nome, nickname, anno)  
...  
>>> foo(**{'nome': 'micky', 'nickname': 'pisi', 'anno': 1985})  
micky pisi 1985
```

Lo spacchettamento è estremamente potente quando è applicato a un parametro `kwargs`, poiché possiamo disinteressarci del nome e del numero sia dei parametri sia degli argomenti:

```
>>> def foo(**kwargs):  
...     for k, v in kwargs.items():  
...         print(k, v)  
...  
>>> foo(**{'nome': 'micky', 'nickname': 'pisi', 'anno': 1985})  
nickname pisi  
nome micky  
anno 1985
```

Concludiamo osservando come le chiavi del dizionario debbano essere delle stringhe:

```
>>> foo(**{'nome': 'micky', 'nickname': 'pisi', 1985: 'anno'})  
Traceback (most recent call last):  
...  
TypeError: foo() keywords must be strings
```

Funzioni anonime

La parola chiave `lambda` utilizzata nella seguente forma:

`lambda par1, par2,..., parN: espressione`

crea una funzione che ha come parametri `par1,..., parN` e che restituisce `espressione`:

```
>>> f = lambda x, y: x + y
>>> f(1, 2)
3
```

A differenza della `def`, la parola chiave `lambda` non viene interpretata come una istruzione, per cui può essere utilizzata all'interno delle espressioni, laddove `def` non può:

```
>>> f = lambda x: x ** 2 if True else None
>>> f(4)
16
```

Abbiamo visto che l'istruzione `def` crea un oggetto di tipo funzione, lo assegna implicitamente all'etichetta interposta tra la `def` e le parentesi tonde e assegna poi una stringa contenente il nome dell'etichetta all'attributo `__name__` della funzione:

```
>>> def foo():
...     pass
...
>>> foo.__name__
'foo'
>>> moo = foo
>>> del foo
>>> moo.__name__
'foo'
```

La parola chiave `lambda` crea una funzione che non viene implicitamente assegnata a una etichetta. Se la si vuole assegnare a una etichetta (ma questo ha poco senso, perché per fare ciò esiste l'istruzione `def`), è necessario farlo in modo esplicito:

```
>>> lambda x, y: x * y # Genera una funzione
```

```
<function <lambda> at 0x7f7b32997710>
>>> f = lambda x, y: x * y # Assegno esplicitamente la funzione a una etichetta
>>> f(4, 5) # Posso utilizzare la funzione tramite l'etichetta
20
```

Poiché una funzione `lambda` non viene implicitamente assegnata a una etichetta, il suo attributo `__name__` fa riferimento al nome generico '`<lambda>`'.

```
>>> f1 = lambda x, y: print(x * y)
>>> f2 = lambda a, b, c: print(min(a, b, c))
>>> f1(10, 11)
110
>>> f2(100, 11, 12)
11
>>> f1.__name__
'<lambda>'
>>> f2.__name__
'<lambda>'
```

Queste funzioni non possono, quindi, essere distinte sulla base del loro nome e per tale motivo vengono chiamate *funzioni anonime*.

Le funzioni anonime supportano tutte le modalità di matching viste in precedenza:

```
>>> f = lambda a, b=2, *varargs, **kwargs: print(a, b, varargs, kwargs)
>>> f(1)
1 2 () {}
>>> f(1, 'due')
1 due () {}
>>> f(1, 'due', 11, 22, 33)
1 due (11, 22, 33) {}
>>> f(1, 'due', 11, 22, 33, c=333, nome='python')
1 due (11, 22, 33) {'c': 333, 'nome': 'python'}
```

compresa quella `keyword-only`:

```
>>> f = lambda *, x, y: x + y
>>> f(y=1, x=2)
3
>>> f(1, 2)
Traceback (most recent call last):
...
TypeError: <lambda>() takes 0 positional arguments but 2 were given
```

Le funzioni anonime, se usate a sproposito, tipicamente rendono il codice poco leggibile. Al contrario, se usate con sapienza consentono di compattarlo aumentandone la leggibilità. Il consiglio è di avvalersi delle `lambda` laddove serve una *piccola* funzione con utilizzo limitato a una espressione, da non

riutilizzare, quindi, in altre parti del programma.

Vediamo un esempio concreto. Supponiamo di avere la seguente lista di sequenze:

```
>>> notable_dates = [(20, 7, 1969), (1666,), (11, 1918), [9, 11, 1989], (25, 11, 1915)]
```

Gli elementi di questa lista sono delle sequenze, ciascuna delle quali rappresenta una data relativa a un evento importante. Vogliamo ordinare la lista sulla base dell'ultimo elemento di ogni sequenza. La soluzione ci pare immediata: utilizziamo il metodo `list.sort()` della lista passando al suo parametro `key` una funzione che prende una sequenza e ne restituisce l'ultimo elemento. Per fare ciò possiamo usare, ad esempio, la classe `operator.itemgetter` della libreria standard, passandole come argomento l'indice dell'elemento che vogliamo restituisca, ovvero `-1`, nel nostro caso:

```
>>> import operator
>>> notable_dates.sort(key=operator.itemgetter(-1))
>>> notable_dates
[(1666,), (25, 11, 1915), (11, 1918), (20, 7, 1969), [9, 11, 1989]]
```

Sembra proprio che `operator.itemgetter` risolva il nostro problema. Ci accorgiamo, tuttavia, che la nostra funzione chiave non può essere utilizzata quando la lista contiene una sequenza vuota:

```
>>> notable_dates = [(20, 7, 1969), (1666,), (11, 1918), [9, 11, 1989], (25, 11, 1915), ()]
>>> notable_dates.sort(key=operator.itemgetter(-1))
Traceback (most recent call last):
...
IndexError: tuple index out of range
```

Noi, però, vorremmo gestire anche questo caso, per cui ci rendiamo conto che `operator.itemgetter` non è la soluzione che fa per noi. Non possiamo fare altro che definire una funzione apposita:

```
>>> def last(seq):
...     return seq[-1] if seq else 0
...
>>> notable_dates = [(20, 7, 1969), (1666,), (11, 1918), [9, 11, 1989], (25, 11, 1915), ()]
>>> notable_dates.sort(key=last)
>>> notable_dates
[(), (1666,), (25, 11, 1915), (11, 1918), (20, 7, 1969), [9, 11, 1989]]
```

Siamo abbastanza soddisfatti, ma non completamente. La nostra funzione

chiave si limita a valutare una singola espressione, e per di più in un solo punto del programma. Forse è proprio il caso in cui l'utilizzo di una lambda è la soluzione migliore, poiché ci permette di avere la definizione della funzione nello stesso punto in cui la si utilizza, compattando così il codice e rendendolo, tra l'altro, più leggibile:

```
>>> notable_dates = [(20, 7, 1969), (1666,), (11, 1918), [9, 11, 1989], (25, 11, 1915), ()]  
>>> notable_dates.sort(key=lambda seq: seq[-1] if seq else 0)  
>>> notable_dates  
[(), (1666,), (25, 11, 1915), (11, 1918), (20, 7, 1969), [9, 11, 1989]]
```

Un altro contesto in cui le lambda si rivelano molto utili è la *programmazione funzionale*, dove è usuale passare le funzioni come argomenti ad altre funzioni e restituirle come valori di ritorno. In questi casi, come possiamo intuire, l'utilizzo delle funzioni anonime risulta estremamente comodo.

NOTA

Se siamo interessati ad approfondire la programmazione funzionale con Python, possiamo consultare la seguente pagina sul sito ufficiale: <http://docs.python.org/3/howto/functional.html>.

Introspezione sulle funzioni

Nel Capitolo 2 abbiamo visto come è possibile fare dell'introspezione sugli oggetti mediante le funzioni built-in `dir()` e `help()`. Abbiamo anche introdotto il modulo `dis`, che consente di disassemblare il codice Python in bytecode. In questa sezione parleremo di tutti gli strumenti che abbiamo a disposizione per effettuare l'introspezione sulle funzioni.

Introspezione tramite gli attributi

Ci sono vari attributi di una funzione che ci consentono di fare dell'introspezione, come si evince dal seguente esempio:

```
>>> def foo(a, b, c=10, *varargs, d=99, **kwargs):
...     """Esempio di docstring"""
...     e = [1, 2, 3]
...     print(f)
...     def moo():
...         pass
...     return moo
...
>>> f = 100
>>> moo = foo(1, 2)
100
>>> foo.__name__ # Nome della funzione
'foo'
>>> foo.__qualname__
'foo'
>>> moo.__name__
'moo'
>>> moo.__qualname__
'foo.<locals>.moo'
>>> foo.__defaults__ # Argomenti di default
(10,)
>>> foo.__kwdefaults__ # Argomenti keyword only di default
{'d': 99}
>>> foo.__module__ # Module che contiene `foo()`
'__main__'
>>> foo.__doc__
'Esempio di docstring'
```

NOTA

Per maggiori dettagli sull'attributo `__qualname__` si veda la PEP-3155.

L'attributo `__code__` di una funzione fa riferimento a un oggetto di tipo `code`:

```
>>> c = foo.__code__
>>> type(c)
<class 'code'>
```

Questo ci consente di avere informazioni sui parametri della funzione, sulle etichette e sulle costanti:

```
>>> c.co_argcount # Numero dei parametri, esclusi *varargs e **kwargs
3
>>> c.co_kwonlyargcount # Numero dei parametri con matching keyword only (solamente `d`)
1
>>> c.co_nlocals # Numero delle etichette locali
7
>>> c.co_varnames # Elenco dei nomi delle etichette locali
('a', 'b', 'c', 'd', 'varargs', 'kwargs', 'e')
>>> c.co_names # Nomi etichette utilizzate ma non definite nella funzione
('print', 'f')
>>> c.co_consts # Elenco costanti (None è restituito implicitamente)
(None, 1, 2, 3)
```

Introspezione statica mediante il modulo `dis`

Consideriamo ancora la funzione dell'esempio precedente:

```
>>> def foo(a, b, c=10, *varargs, d=99, **kwargs):
...     """Esempio di docstring"""
...     e = [1, 2, 3]
...     print(f)
...     def moo():
...         pass
...     return moo
```

Possiamo ottenere delle informazioni su di essa anche mediante il modulo `dis` della libreria standard:

```
>>> import dis
>>> dis.show_code(foo)

Name: foo
Filename: <stdin>
Argument count: 3
Kw-only arguments: 1
```

Number of locals: 7

Stack size: 3

Flags: OPTIMIZED, NEWLOCALS, VARARGS, VARKEYWORDS, NOFREE

Constants:

0: None

1: 1

2: 2

3: 3

Names:

0: print

1: f

Variable names:

0: a

1: b

2: c

3: d

4: varargs

5: kwargs

6: e

Come sappiamo, questo modulo ci consente anche di vedere il bytecode generato:

```
>>> import dis
>>> dis.dis(foo)

2 0 LOAD_CONST 1 (1)
3 LOAD_CONST 2 (2)
6 LOAD_CONST 3 (3)
9 BUILD_LIST 3
12 STORE_FAST 6 (e)
3 15 LOAD_GLOBAL 0 (print)
18 LOAD_GLOBAL 1 (f)
21 CALL_FUNCTION 1 (1 positional, 0 keyword pair)
24 POP_TOP
25 LOAD_CONST 0 (None)
28 RETURN_VALUE
```

La funzione `dis.dis()` accetta come argomento anche una stringa:

```
>>> dis.dis('def foo(): pass')

1 0 LOAD_CONST 0 (<code object foo at ..., line 1>)
3 LOAD_CONST 1 ('foo')
6 MAKE_FUNCTION 0
9 STORE_NAME 0 (foo)
12 LOAD_CONST 2 (None)
15 RETURN_VALUE
```

```

>>> s = ""
... a = 33
... b = [1, 2, 3]
... b.append(4)
... ""
>>> dis.dis(s)
2 0 LOAD_CONST 0 (33)
3 STORE_NAME 0 (a)
3 6 LOAD_CONST 1 (1)
9 LOAD_CONST 2 (2)
12 LOAD_CONST 3 (3)
15 BUILD_LIST 3
18 STORE_NAME 1 (b)
4 21 LOAD_NAME 1 (b)
24 LOAD_ATTR 2 (append)
27 LOAD_CONST 4 (4)
30 CALL_FUNCTION 1 (1 positional, 0 keyword pair)
33 POP_TOP
34 LOAD_CONST 5 (None)
37 RETURN_VALUE

```

Introspezione con le annotazioni

Le *annotazioni* (*function annotations*) sono dei *metadati* opzionali relativi ai parametri e al valore di ritorno di una funzione. La sintassi adottata per inserire un'annotazione su un parametro consiste nel far seguire il parametro da un delimitatore di due punti seguito, a sua volta, da una espressione:

```

>>> def foo(a: list, b: int):
...     return a * b
...
>>> foo([1, 2, 3], 2)
[1, 2, 3, 1, 2, 3]

```

La sintassi adottata per inserire l'annotazione sul valore di ritorno consiste, invece, nel far seguire la parentesi tonda di chiusura da una freccia, a sua volta seguita da una espressione:

```

>>> def foo(a: list, b: int) -> 'a * b':
...     return a * b

```

Le annotazioni devono essere delle espressioni valide. Ecco, infatti, cosa accade se come annotazione sul valore di ritorno indichiamo `a * b` anziché `'a * b'`:

```
>>> def foo(a: list, b: int) -> a * b:
...     return a * b
...
Traceback (most recent call last):
...
NameError: name 'a' is not defined
```

Le annotazioni vengono assegnate all'attributo `__annotations__` della funzione:

```
>>> def foo(a: list, b: int, c: 'python') -> None:
...     pass
...
>>> foo.__annotations__
{'a': <class 'list'>, 'b': <class 'int'>, 'c': 'python', 'return': None}
```

Ciò che viene assegnato all'annotazione è il risultato dell'espressione:

```
>>> def foo(a: max(100, 2, 200, 1)):
...     pass
...
>>> foo.__annotations__
{'a': 200}
```

I valori di default vanno indicati dopo l'annotazione:

```
>>> def foo(a: int = 100):
...     print(a)
...
>>> foo.__annotations__
{'a': <class 'int'>}
>>> foo()
100
```

Le annotazioni hanno diversi ambiti di utilizzo. Il più immediato è quello legato alla documentazione del codice. A differenza delle docstring, che ci consentono di ottenere la documentazione della funzione, le annotazioni sono più specifiche e permettono di effettuare dell'introspezione direttamente sui parametri e sul valore di ritorno:

```
>>> def newton(m: 'Massa in kg', a: 'Accelerazione in m/s^2') -> 'Forza in N':
...     return m * a
...
>>> newton.__annotations__['m']
'Massa in kg'
>>> newton.__annotations__['return']
'Forza in N'
>>> newton(200, 0.75)
```

Possono essere utilizzate dagli IDE per mostrare il tipo che ci si aspetta per gli argomenti e per il valore di ritorno, oppure possono essere utilizzate per effettuare il *type checking* degli argomenti e altro ancora.

NOTA

Esiste una variante di Python, chiamata *mypy*, che fa uso delle function annotation per combinare assieme i vantaggi della tipizzazione statica e dinamica. Consideriamo, ad esempio, il seguente file:

```
$ cat foo.py
def foo(a: str):
    print(a)
```

```
foo(100)
```

Se lo eseguiamo con Python, viene stampato 100:

```
$ python foo.py
100
```

Se invece lo eseguiamo con *mypy*, viene rilevato un errore:

```
$ mypy foo.py
foo.py, line 4: Argument 1 to "foo" has incompatible type "int"
```

Ovviamente il type-checking non si ottiene gratis e bisogna fare i conti con le prestazioni:

```
$ time python foo.py
100
```

```
real 0m0.038s
user 0m0.028s
sys 0m0.000s
```

```
$ time mypy foo.py
foo.py, line 4: Argument 1 to "foo" has incompatible type "int"
```

```
real 0m0.427s
user 0m0.400s
sys 0m0.024s
```

Per maggiori informazioni su mypy si può consultare il sito ufficiale:
<http://www.mypy-lang.org/>.

Concludiamo questa sezione osservando che le annotazioni non possono essere utilizzate con le funzioni anonime:

```
>>> lambda x, y: x / y
<function <lambda> at 0x7f3e79f59560>
>>> lambda x, y: 'numero positivo': x / y
File "<stdin>", line 1
lambda x, y: 'numero positivo': x / y
^
SyntaxError: invalid syntax
```

Per maggiori informazioni sulle annotazioni, possiamo consultare la PEP-3107.

Introspezione mediante il modulo inspect

Consideriamo il seguente modulo `foo`:

```
$ cat foo.py
# Definiamo una funzione `foo()` sulla quale faremo dell'introspezione
def foo(a, b, c=10, *varargs, d=99, **kwargs):
    """Fai qualcosa di apparentemente poco utile..."""
    print(e) # Stampa l'etichetta libera `e`
    f = [1, 2]
```

Con il modulo `inspect` della libreria standard possiamo scoprire varie cose su un oggetto:

```
>>> import inspect
>>> from foo import foo
>>> inspect.ismodule(foo)
False
>>> inspect.ismethod(foo)
False
>>> inspect.isfunction(foo)
True
>>> c = foo.__code__
>>> inspect.iscode(c)
True
```

Possiamo ottenere la `docstring` e i commenti:

```
>>> inspect.getdoc(foo)
'Fai qualcosa di apparentemente poco utile...'
```



```
>>> inspect.getcomments(foo) # Restituisce i commenti che precedono la definizione
'# Definiamo una funzione `foo()` sulla quale faremo introspezione\n'
>>> inspect.getsourcefile(foo)
'./foo.py'
```

Possiamo sapere se un oggetto è built-in:

```
>>> inspect.isbuiltin(sum)
True
>>> import sys
>>> inspect.isbuiltin(sys)
False
```

Con il modulo `inspect` possiamo anche ottenere una stringa contenente il testo del codice sorgente di un oggetto, se questo è definito in un file:

```
$ cat myfile.py
def foo(a, b):
    print(a, b)
$ python -c "import inspect, myfile; print(inspect.getsource(myfile.foo))"
def foo(a, b):
    print(a, b)
```

Se possiamo ottenere il codice sorgente di un oggetto, allora possiamo anche modificarlo ed eseguirlo modificato:

```
>>> import inspect, myfile
>>> source = inspect.getsource(myfile.foo)
>>> print(source)
def foo(a, b):
    print(a, b)
>>> newsource = source.replace("print(a, b)", "print(a, b, sep=' | ')")
>>> exec(source)
>>> foo(1, 2)
1 2
>>> exec(newsource)
>>> foo(1, 2)
1 | 2
```

La funzione `inspect.signature()` restituisce la signature dell'oggetto callable passato come argomento:

```
>>> def foo(a: 'annotazione inutile', b=33, *args) -> str:
...     print(a, b, args)
...     return 'python3'
...
>>> import inspect
>>> sig = inspect.signature(foo)
>>> type(sig)
```

```
<class 'inspect.Signature'>
>>> print(sig)
(a:'annotazione inutile', b=33, *args) -> str
```

La signature ci consente di ottenere informazioni sui parametri della funzione:

```
>>> sig.parameters['a'].annotation
'annotazione inutile'
>>> sig.parameters['b'].default
33
```

Per maggiori informazioni sul modulo `inspect`, possiamo consultare la documentazione online, alla pagina <http://docs.python.org/3/library/inspect.html>.

Introspezione con le funzioni built-in

Oltre che con le funzioni built-in `dir()` e `help()`, l'introspezione può essere effettuata anche tramite la classe built-in `type` e le funzioni built-in `id()`, `getattr()`, `hasattr()` e `locals()`. Abbiamo già incontrato diverse volte le prime due. Le funzioni `getattr()`, `hasattr()` ci consentono, rispettivamente, di ottenere un attributo di un oggetto e di sapere se un oggetto ha un certo attributo, dato il nome di quest'ultimo come stringa:

```
>>> def foo(a, b):
...     c = 33
...     print(a, b, c)
...
>>> hasattr(foo, 'a')
False
>>> hasattr(foo, 'c')
False
>>> foo.c = 44
>>> hasattr(foo, 'c')
True
>>> getattr(foo, 'c')
44
>>> foo(1, 2)
1 2 33
>>> hasattr(foo, '__module__')
True
>>> getattr(foo, '__module__')
'__main__'
```

La funzione `locals()` restituisce il namespace corrente sotto forma di dizionario:

```
>>> def foo(a, b, c=10, *varargs, d=99, **kwargs):
```

```
... print('locals() inizio corpo: ', locals())
... print(f)
... e = [1, 2]
... print('locals() fine corpo: ', locals())
...
>>> f = 10
>>> foo(1, 2)
locals() inizio corpo: {'kwargs': {}, 'd': 99, 'c': 10, 'varargs': (), 'a': 1, 'b': 2}
10
locals() fine corpo: {'kwargs': {}, 'e': [1, 2], 'd': 99, 'c': 10, 'varargs': (), 'a': 1, 'b': 2}
```

Se il concetto di namespace non vi è chiaro, non preoccupatevi, lo vedremo in dettaglio nel Capitolo 4.

Generatori

In questa sezione parleremo in modo abbastanza dettagliato dei generatori. Poiché è un argomento avanzato, possiamo prenderci la licenza di saltare qualche parte, per poi eventualmente riprenderla all'occorrenza.

Funzioni generatore

Nel Capitolo 1 abbiamo detto che il protocollo di iterazione consente di recuperare, uno per volta, gli elementi da un oggetto iterabile, evitando così che questo venga interamente caricato in memoria. Consideriamo, ad esempio, la classe `range`. Quando la si utilizza in un contesto di iterazione, come nel seguente caso:

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

il ciclo `for` non carica la sequenza di numeri interi in memoria, ma compie una serie di passi, in modo da recuperare gli elementi uno per volta. Come primo passo, crea un oggetto iteratore:

```
>>> i = iter(range(5)) # Crea un iteratore
```

Dopodiché chiama il metodo `i.__next__()` per ottenere gli elementi uno per volta:

```
>>> i.__next__()
0
>>> i.__next__()
1
>>> i.__next__()
2
>>> i.__next__()
3
>>> i.__next__()
4
```

Quando l'iteratore non ha più elementi da restituire, lo comunica al ciclo `for` lanciando una eccezione di tipo `StopIteration`:

```
>>> i.__next__()  
Traceback (most recent call last):  
...  
StopIteration
```

In tantissimi casi gli oggetti sui quali si va a iterare vengono costruiti e restituiti da qualche funzione, come nel seguente esempio:

```
>>> def wondrous(n):  
...     """Restituisci una lista di numeri HOTPO, a partire da n."""  
...     mylist = []  
...     while n != 1:  
...         mylist.append(n)  
...         n = n // 2 if n % 2 == 0 else 3*n + 1  
...     else:  
...         mylist.append(1)  
...     return mylist  
>>> for i in wondrous(5):  
...     print(i)  
...  
5  
16  
8  
4  
2  
1
```

La funzione `wondrous()` prende un numero intero positivo n e sulla base di questo crea e restituisce una lista:

```
>>> wondrous(5)  
[5, 16, 8, 4, 2, 1]  
>>> wondrous(13)  
[13, 40, 20, 10, 5, 16, 8, 4, 2, 1]  
>>> wondrous(29)  
[29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Questa lista ha come primo elemento n . Il secondo elemento è pari a $3*n + 1$ se n è dispari, altrimenti è pari a $n // 2$. Quando si arriva a 1 la lista è completa. La dimensione della lista non è conosciuta a priori, poiché dipende dal valore di n .

NOTA

Nel 1937 Lothar Collatz disse che, qualunque sia il numero n intero positivo, questo algoritmo giunge sempre a termine. A oggi questa affermazione, detta *congettura di Collatz*, non è dimostrata, anche se la maggior parte dei matematici che se ne sono occupati pensa che sia vera.

Quando la funzione viene utilizzata in un ciclo `for`, crea la lista di tutti gli elementi e poi la restituisce in modo che si iteri su di essa:

```
>>> n = int(input("Inserisci un numero intero positivo: "))
Inserisci un numero intero positivo: 29
>>> for i in wondrous(n):
...     print(i, end=' ')
...     if n // i == 2:
...         break
...
29 88 44 22 11
```

C'è stato un dispendio di risorse inutile, poiché abbiamo caricato in memoria una lista di 19 elementi:

```
>>> w = wondrous(29)
>>> w
[29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
>>> len(w)
19
```

ma ne abbiamo utilizzati solo cinque. Le risorse inutili sono state spese sia dal punto di vista computazionale, perché la funzione ha effettuato tutti i calcoli per generare la lista intera, sia dal punto di vista della memoria, perché la lista è stata interamente caricata in RAM. In questo esempio il dispendio di risorse è minimo, poiché la lista contiene solamente 19 elementi, ma è evidente che in generale questo approccio è da evitare.

Se la funzione non avesse restituito tutta la sequenza, ma avesse invece *prodotto* (generato) un elemento per volta su richiesta, avremmo evitato questo dispendio di risorse. Possiamo ottenere questo comportamento definendo un particolare tipo di funzione, detta *funzione generatore*.

NOTA

Nei testi in lingua inglese, una funzione generatore è chiamata semplicemente *generator*.

Dal punto di vista della sintassi, una funzione generatore si distingue dalle funzioni che abbiamo visto sinora per il fatto che il controllo al chiamante viene restituito con l'espressione `yield item` piuttosto che con una istruzione `return`:

```
>>> def foo(seq):
...     print("Inizia l'esecuzione del codice...")
...     for item in seq:
...         yield item
...     print(item)
```

Questa funzione restituisce un iteratore di tipo `generator`, detto *generatore*. Come possiamo vedere, la chiamata alla funzione non causa l'esecuzione del codice, ma crea un generatore e lo restituisce:

```
>>> g = foo('abc')
>>> type(g)
<class 'generator'>
```

L'esecuzione del codice della funzione inizia con la chiamata al metodo `g.__next__()` del generatore:

```
>>> g.__next__()
Inizia l'esecuzione del codice...
'a'
```

Questa causa l'esecuzione della `print()`, l'assegnamento a `item` del primo elemento della sequenza e la restituzione di `item` al chiamante con l'espressione `yield item`. A questo punto l'esecuzione viene *sospesa*, e il namespace locale viene salvato in modo da poter essere riutilizzato quando la funzione verrà *risvegliata*, il che accadrà con la successiva chiamata a `g.__next__()`.

NOTA

Una importante differenza tra le funzioni normali e le funzioni generatore consiste nel fatto che il namespace locale delle prime viene creato al momento della chiamata e cancellato quando la funzione restituisce il controllo al chiamante, mentre il namespace delle seconde non viene cancellato, perché deve essere riutilizzato quando si chiede al generatore l'elemento successivo, in modo tale che l'esecuzione riprenda nelle stesse

condizioni in cui era stata interrotta.

La richiesta dell'elemento successivo viene fatta, ancora una volta, con la chiamata a `g.__next__()`. Con questa chiamata il generatore riprende l'esecuzione del codice a partire dall'istruzione successiva alla `yield item`, nelle stesse condizioni in cui si era interrotta, quindi con lo stesso namespace di prima. Viene così stampato `item` (la stringa 'a'), e poi viene assegnato ad `item` l'elemento successivo (la stringa 'b'), restituito poi al chiamante con l'espressione `yield item`:

```
>>> g.__next__()
a
'b'
```

Con la successiva chiamata a `g.__next__()` l'esecuzione riprende ancora dalla stampa di `item` (la stringa 'b'), poi prosegue con l'assegnamento dell'elemento successivo della sequenza (la stringa 'c') a `item`, e si conclude con la restituzione di `item` al chiamante:

```
>>> g.__next__()
b
'c'
```

Infine, l'ultima chiamata a `g.__next__()` causa la stampa di `item` e l'uscita dal ciclo `for`. La funzione, quindi, termina definitivamente la sua esecuzione, lanciando una eccezione di tipo `StopIteration`:

```
>>> g.__next__()
c
Traceback (most recent call last):
...
StopIteration
```

Come abbiamo potuto constatare, il comportamento di una funzione generatore è molto differente da quello di una funzione normale:

- per una *funzione normale* l'esecuzione del codice avviene al momento della chiamata e termina con la restituzione esplicita (con `return`) o implicita di un risultato, e ogni chiamata ha il suo namespace locale;
- per una *funzione generatore* l'esecuzione del codice non avviene al momento della chiamata, la quale ha il solo effetto di creare e restituire un oggetto generatore, che indichiamo con `g`. L'esecuzione del codice avviene successivamente, su richiesta, tramite le chiamate al metodo `g.__next__()`. La

funzione non ha il compito di restituire un solo risultato, ma *una serie* di risultati, ognuno dei quali viene prodotto con la chiamata a `g.__next__()`. Quando tutti i risultati sono stati prodotti, `g.__next__()` non ha più risultati da produrre, per cui porta a termine l'esecuzione del codice della funzione e poi lancia un'eccezione di tipo `StopIteration`.

Tornando alla funzione iniziale `wondrous()`, adesso sappiamo che possiamo implementarla come funzione generatore, nel seguente modo:

```
>>> def wondrous(n):
...     """Generatore di numeri HOTPO, a partire da n."""
...     while n != 1:
...         yield n
...         n = n // 2 if n % 2 == 0 else 3 * n + 1
...     else:
...         yield 1
```

Quando la funzione viene chiamata, restituisce un generatore, tramite il quale possiamo produrre gli elementi della sequenza uno alla volta:

```
>>> g = wondrous(5)
>>> g.__next__()
5
>>> g.__next__()
16
>>> g.__next__()
8
>>> g.__next__()
4
>>> g.__next__()
2
>>> g.__next__()
1
>>> g.__next__()
Traceback (most recent call last):
...
StopIteration
```

A ogni richiesta il generatore produce e restituisce l'elemento corrispondente, il che significa che l'elemento i -esimo viene prodotto solo in corrispondenza della richiesta i -esima e non prima.

Come abbiamo detto, un generatore è un iteratore, per cui l'istruzione `for` ci consente di iterare in modo automatico su di esso:

```
>>> for i in wondrous(5):
...     print(i)
...
5
16
8
4
2
1
```

Se vogliamo saltare una parte iniziale degli elementi del generatore, possiamo utilizzare la classe `itertools.dropwhile`:

```
>>> list(wondrous(23))
[23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
>>> from itertools import dropwhile
>>> for i in dropwhile(lambda num: num > 4, wondrous(23)):
...     print(i)
...
4
2
1
```

Come possiamo osservare, sono stati saltati tutti gli elementi sino agli ultimi tre. Infatti la classe `itertools.dropwhile` prende come primo argomento una funzione `fun` e come secondo argomento un iteratore `iterator`, e restituisce un altro iteratore. Quest'ultimo produce solamente i primi elementi `i` di `iterator`, per i quali `fun(i)` restituisce `True`:

```
>>> for i in dropwhile(lambda num: num > 8, wondrous(23)):
...     print(i)
...
5
16
8
4
2
1
```

Nell'esempio appena visto, il primo elemento `num` per il quale `num > 8` risulta `False` è 5.

Se, invece, vogliamo saltare un dato intervallo di elementi, allora possiamo utilizzare la classe `itertools.islice`:

```
>>> from itertools import islice
>>> list(wondrous(23))
[23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

```
>>> for i in islice(wondrous(23), 10, None):
```

```
... print(i)
```

```
...
```

```
5
```

```
16
```

```
8
```

```
4
```

```
2
```

```
1
```

```
>>> for i in islice(wondrous(23), 0, 2):
```

```
... print(i)
```

```
...
```

```
23
```

```
70
```

Il primo elemento di `itertools.islice` è l'iteratore, il secondo l'indice dell'elemento iniziale e il terzo l'indice di quello finale.

Per maggiori informazioni sulle funzioni generatore, possiamo consultare la PEP-0255.

Coroutine

Le *coroutine* sono delle particolari funzioni, che, come le funzioni generatore, sospendono la loro esecuzione per poi riprenderla su richiesta del chiamante. A differenza di queste ultime, però, non producono un valore, bensì lo *consumano*. Ciò significa che la coroutine avvia l'esecuzione del codice quando le viene passato un oggetto, dopodiché lo consuma e poi sospende la sua attività finché non viene risvegliata passandole un nuovo oggetto da consumare.

Come per le funzioni generatore, per creare una coroutine si utilizza l'espressione `yield`, ma questa volta alla destra dell'operatore di assegnamento, nella forma `item = (yield)`:

```
>>> def foo():
```

```
... print("Inizia l'esecuzione del codice...")
```

```
... while True:
```

```
... x = (yield)
```

```
... print(x)
```

```
... print("Termina l'esecuzione del codice...")
```

La chiamata alla funzione non causa l'esecuzione del codice, ma restituisce un generatore:

```
>>> g = foo()
>>> type(g)
<class 'generator'>
```

Per avviare l'esecuzione del codice si utilizza il metodo `g.__next__()`. Questo esegue la `print()`, entra nella suite dell'istruzione `while` e sospende l'esecuzione in corrispondenza della `yield`. Possiamo riprendere l'esecuzione della funzione in due modi: con il metodo `g.send()` oppure con `g.__next__()`. Entrambi i metodi riprendono l'esecuzione del codice fino alla prossima `yield`, per poi sospenderla nuovamente. La differenza tra i metodi `g.send()` e `g.__next__()` risiede nel fatto che, con il primo, è possibile passare alla funzione l'oggetto da consumare, che verrà assegnato a `x`, mentre con il secondo questo non si può fare:

```
>>> g.send('python')
python
>>> g.__next__() # Non viene passato nulla, per cui x = None
None
>>> g.send(1)
1
>>> g.send(['a', 'b', 'c'])
['a', 'b', 'c']
>>> g.__next__()
None
```

Il metodo `g.close()` interrompe definitivamente l'esecuzione a partire dal punto in cui ci si trova, per cui, se la chiamiamo, la `print()` che si trova al di fuori del blocco `while` non viene eseguita:

```
>>> g.close()
>>> g.__next__()
Traceback (most recent call last):
...
StopIteration
>>> g.send(3)
Traceback (most recent call last):
...
StopIteration
```

Quando l'esecuzione della funzione termina senza aver chiamato `g.close()`, allora viene lanciata una eccezione di tipo `StopIteration`:

```
>>> def foo():
...     print("Inizia l'esecuzione del codice...")
...     x = True
...     while x != 'python':
...         x = (yield)
```

```

... print(x)
... print("Termina l'esecuzione del codice...")
...
>>> g = foo()
>>> g.__next__()
Inizia l'esecuzione del codice...
>>> g.__next__()
None
>>> g.send('abc')
abc
>>> g.send('python')
python
Termina l'esecuzione del codice...
Traceback (most recent call last):
...
StopIteration

```

Una funzione può essere allo stesso tempo sia coroutine che generatore:

```

>>> def foo(seq):
...     for item in seq:
...         x = (yield item)
...         print(x)

```

Vediamo cosa accade in questo caso. Con la prima chiamata al metodo `g.__next__()` viene eseguito il codice sino alla `yield item` (la quale produce `item`) e poi l'esecuzione viene nuovamente interrotta:

```

>>> g = foo('abc')
>>> g.__next__()
'a'

```

Come abbiamo visto in precedenza, la funzione può essere *risvegliata* sia con `g.send()` che con `g.__next__()`. Vediamo cosa accade con `g.__next__()`:

```

>>> g.__next__()
None
'b'

```

È stato assegnato `None` a `x`, poi l'esecuzione ha proseguito sino alla `yield item` (la quale ha prodotto `item`), dopodiché è stata nuovamente interrotta. Se ora la risvegliamo con `g.send()`, l'unica differenza sarà che verrà assegnato a `x` l'oggetto argomento di `g.send()`:

```

>>> g.send('ciao')
ciao

```

'c'

La funzione non può più produrre risultati, per cui un'altra chiamata a `g.send()` o `g.__next__()` darà luogo al lancio di una eccezione `StopIteration`:

```
>>> g.send(55)
55
Traceback (most recent call last):
...
StopIteration
```

Per maggiori informazioni sulle *coroutine*, possiamo consultare la PEP-0342.

Generator expression

Nel Capitolo 2 abbiamo visto come creare liste, dizionari o insiemi in modo conciso, mediante le comprehension. In questo capitolo abbiamo visto, invece, come produrre degli elementi di una collezione uno alla volta su richiesta, mediante le funzioni generatore. Le *generator expression* sono delle espressioni che consentono di combinare assieme le due cose, permettendo di creare un generatore in modo conciso, con una sintassi analoga a quella utilizzata per le comprehension, che fa uso, questa volta, delle parentesi tonde:

```
>>> g = (x ** 2 for x in range(10) if x % 2 == 0)
>>> type(g)
<class 'generator'>
>>> g.__next__()
0
>>> g.__next__()
4
>>> g.__next__()
16
>>> g.__next__()
36
>>> g.__next__()
64
>>> g.__next__()
Traceback (most recent call last):
...
StopIteration
```

Quindi, se volessimo iterare su una sequenza di questo tipo, piuttosto che utilizzare una list comprehension utilizziamo una generator expression:

```
>>> for i in (x ** 2 for x in range(10) if x % 2 == 0):
...     print(i, end=' ')
...
```

Possiamo rimpiazzare le comprehension con le generator expression anche dove utilizzavamo le prime come argomenti di una funzione. Il seguente esempio:

```
>>> [x ** 2 for x in range(6) if x % 2 != 0]
[1, 9, 25]
>>> sum([x ** 2 for x in range(6) if x % 2 != 0], 10) # Aggiungi 10 alla somma
45
```

può essere riscritto con una generator expression:

```
>>> sum((x ** 2 for x in range(6) if x % 2 != 0), 10)
45
```

Se la funzione non ha altri argomenti, possiamo anche omettere le parentesi tonde della generator expression:

```
>>> sum(x ** 2 for x in range(6) if x % 2 != 0)
35
```

Per maggiori informazioni sulle generator expression, possiamo consultare la PEP-0289.

Sub-generatori

Consideriamo le seguenti funzioni generatrici, rispettivamente, di numeri di Fibonacci e HTPO:

```
>>> def fibonacci(n, start=1):
...     """Genera numeri di fibonacci"""
...     a, b = start, start + 1
...     while a < n:
...         yield a
...         a, b = b, a + b
...
>>> def wondrous(n):
...     """Genera numeri HTPO"""
...     while n != 1:
...         yield n
...         n = n // 2 if n % 2 == 0 else 3 * n + 1
...     else:
```

```
... yield 1
```

Supponiamo di voler generare, per ogni numero di Fibonacci, i corrispondenti numeri HTPO:

```
>>> list(fibonacci(8))
[1, 2, 3, 5]
>>> list(wondrous(1)) # Numeri HTPO corrispondenti al primo numero di `fibonacci(8)`
[1]
>>> list(wondrous(2)) # Numeri HTPO corrispondenti al secondo numero di `fibonacci(8)`
[2, 1]
>>> list(wondrous(3)) # Numeri HTPO corrispondenti al terzo numero di `fibonacci(8)`
[3, 10, 5, 16, 8, 4, 2, 1]
>>> list(wondrous(5)) # Numeri HTPO corrispondenti al quarto numero di `fibonacci(8)`
[5, 16, 8, 4, 2, 1]
```

Vorremmo definire una funzione generatore `wondrous_of_fibonacci()` che faccia questo. Questa funzione per ogni numero di Fibonacci dovrebbe fare delle richieste alla funzione generatore `wondrous()`. Lasciando perdere la gestione degli errori, dovremmo scrivere qualcosa del genere:

```
>>> def wondrous_of_fibonacci(n, start=1):
...     a, b = start, start + 1
...     while a < n:
...         g = wondrous(a)
...         for item in g:
...             yield item
...         a, b = b, a + b
...
>>> list(wondrous_of_fibonacci(8))
[1, 2, 1, 3, 10, 5, 16, 8, 4, 2, 1, 5, 16, 8, 4, 2, 1]
```

La gestione del generatore di numeri HTPO può essere fatta in modo automatico con l'utilizzo dell'espressione `yield from`, la quale sostanzialmente si occupa di iterare sul generatore:

```
>>> def wondrous_of_fibonacci(n, start=1):
...     a, b = start, start + 1
...     while a < n:
...         yield from wondrous(a)
...         a, b = b, a + b
...
>>> list(wondrous_of_fibonacci(8))
[1, 2, 1, 3, 10, 5, 16, 8, 4, 2, 1, 5, 16, 8, 4, 2, 1]
```


Per maggiori informazioni sui sub-generatori, possiamo consultare la PEP-0380.

File

Nei capitoli precedenti abbiamo utilizzato la funzione built-in `open()` in diverse occasioni, principalmente per leggere del testo da file. È giunto il momento di approfondire il discorso.

Lettura e scrittura

La funzione built-in `open()` restituisce un oggetto, detto *file object*, tramite il quale possiamo interagire con un file. È usata principalmente con due argomenti. Il primo, obbligatorio, è una stringa di testo contenente il nome del file (compreso il percorso, se il file non si trova nella directory corrente), mentre il secondo, opzionale, è una stringa che descrive la modalità con la quale il file deve essere aperto.

NOTA

Come vedremo nella sezione *I parametri della funzione open()*, il primo argomento della funzione `open()` può anche essere un *file descriptor*.

Con la modalità `w` il file viene aperto in scrittura (*writing mode*):

```
>>> file = open('myfile', 'w')
>>> file.mode
'w'
>>> file.readable()
False
```

Con questa modalità di apertura, se il file esiste viene sovrascritto, altrimenti viene creato. Per default, se non viene passato il secondo argomento, il file viene aperto in lettura (modalità `r`, *reading mode*):

```
>>> file = open('/etc/hostname')
>>> file.mode
'r'
>>> file.readable(), file.writable()
(True, False)
```

I metodi `file.write()` e `file.read()` ci consentono, rispettivamente, di scrivere e leggere

il contenuto del file. Il metodo `file.write()` prende una stringa come argomento, mentre il metodo `file.read()` restituisce una stringa. Se non viene specificato diversamente, le stringhe sono di testo (modalità `t`, *text*), e in questo caso diciamo che il file object fa riferimento a un file di testo:

```
>>> file.read()
'buttu-oac\n'
```

Facendo seguire alla modalità di apertura il carattere `b`, il file viene aperto in modalità binaria (`b`, *binary mode*) e diciamo che l'object file si riferisce a un file binario. In questo caso le stringhe in lettura o scrittura saranno di byte:

```
>>> file = open('/etc/hostname', 'rb')
>>> file.read()
b'buttu-oac\n'
```

Il metodo `file.write()` restituisce il numero di caratteri o byte scritti sul file, a seconda che si tratti di un file di testo o di un file binario:

```
>>> file = open('myfile', 'w')
>>> f.write('Python 3') # Scriviamo una stringa di testo sul file
8
>>> file = open('myfile', 'wb')
>>> file.write(b'Python 3') # Scriviamo una stringa di byte sul file
8
```

Il metodo `f.read()` per default restituisce tutto il contenuto del file. Ha però un secondo argomento opzionale, che indica il numero di caratteri o byte da leggere:

```
>>> file = open('/etc/hostname')
>>> file.read(5) # Leggiamo cinque caratteri dal file
'buttu'
>>> file = open('/etc/hostname', 'rb')
>>> file.read(6) # Leggiamo sei byte dal file
b'buttu-'
```

Come vedremo tra breve, per default la funzione `open()` effettua il *buffering* in scrittura. Questo significa che, quando scriviamo una stringa sul file, questa non viene scritta immediatamente sul file system ma viene tenuta in memoria. È necessario svuotare il buffer affinché la scrittura sul disco sia effettiva:

```
>>> f = open('myfile', 'w')
>>> f.write('The Pythonic Way - ')
19
```

```
>>> open('myfile').read() # La stringa è ancora nel buffer...
''
```

Il buffer può essere svuotato esplicitamente con il metodo `f.flush()`, oppure in modo implicito al verificarsi di altri eventi, come la chiusura del file:

```
>>> f.flush() # Svuotiamo il buffer in modo esplicito
>>> open('myfile').read()
'The Pythonic Way - '
>>> f.write('Prima edizione')
14
>>> open('myfile').read() # La stringa è ancora nel buffer...
'The Pythonic Way - '
>>> f.close() # Anche la chiusura comporta lo svuotamento del buffer
>>> open('myfile').read()
'The Pythonic Way - Prima edizione'
```

Quando un file object non ha più etichette che vi fanno riferimento, Python lo chiude e libera la memoria da esso occupata:

```
>>> f = open('myfile', 'w')
>>> f.write('The Pythonic Way - Prima edizione')
33
>>> f = 100 # Non vi sono più etichette che fanno riferimento al file object
>>> open('myfile').read() # Infatti il buffer è stato svuotato
'The Pythonic Way - Prima edizione'
```

Quando si raggiunge la fine del file, `f.read()` restituisce una stringa vuota:

```
>>> f = open('myfile')
>>> f.read()
'The Pythonic Way - Prima edizione'
>>> f.read()
''
```

La posizione in cui ci si trova nel file può essere impostata con `f.seek()`. Questo metodo, oltre a impostare la posizione, svuota il buffer e restituisce la posizione impostata:

```
>>> f.seek(0) # Ci riposizioniamo all'inizio del file
0
>>> f.read()
'The Pythonic Way - Prima edizione'
```

Per conoscere la posizione attuale si utilizza il metodo `f.tell()`, il quale, come `f.seek()`, svuota il buffer quando viene chiamato. Entrambi i metodi misurano la posizione in numero di byte e non di caratteri:

```
>>> f = open('myfile', 'w')
>>> f.tell() # La posizione iniziale è 0
0
>>> f.write('a') # Scriviamo un carattere di 1 byte
1
>>> f.tell() # Ci troviamo in posizione 1
1
>>> f.write('è') # Scriviamo un carattere, ma in UTF-8 sono 2 bytes
1
>>> f.tell() # Infatti ci troviamo in posizione 3
3
>>> open('myfile').read() # Il buffer è stato svuotato da tell()
'aè'
```

Se adesso scriviamo partendo dal secondo byte, andiamo a separare i due byte che rappresentano il carattere è:

```
>>> f.seek(2) # Ci posizioniamo dopo il primo byte del carattere `è`
2
>>> f.write('b')
1
>>> f.flush()
>>> open('myfile').read()
Traceback (most recent call last):
...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 1: invalid continuation byte
```

Il metodo `f.seek()` prende anche un secondo argomento opzionale, utilizzato per indicare il riferimento, ovvero la posizione dalla quale deve essere applicato l'*offset*. Il riferimento può assumere solo tre valori: 0, 1 o 2. Il valore 0 è quello di default, e imposta il riferimento all'inizio del file. Il valore 1 usa come riferimento la posizione attuale, mentre il valore 2 usa la fine del file. Al posto dei valori numerici 0, 1 e 2, si possono usare, rispettivamente, le costanti `SEEK_SET`, `SEEK_CUR` e `SEEK_END` del modulo `io`. Nei file di testo è ammesso solo il riferimento all'inizio del file (valore 0, quello di default), oppure alla fine 2 ma con offset pari a 0:

```
>>> f = open('myfile', 'w+') # Il '+' aggiunge anche la modalità lettura `r`
>>> f.readable(), f.writable() # Modalità `w+`: writing + reading
(True, True)
>>> f.write("".join(str(i) for i in range(10)))
10
>>> f.seek(0) # Svuota il buffer e si posiziona all'inizio del file
0
>>> f.read()
'0123456789'
>>> f.seek(5) # Si posiziona dopo il quinto byte
5
>>> f.read(1)
'5'
```

```
>>> f.seek(0, 2) # Ultima posizione
10
>>> f.seek(-3, 2) # Prima del terzultimo byte: non consentito
Traceback (most recent call last):
...
io.UnsupportedOperation: can't do nonzero end-relative seeks
```

Il metodo `f.writelines()` prende come argomento un oggetto iterabile composto da stringhe, e scrive tali stringhe sul file. Il terminatore di linea non viene aggiunto alle stringhe, per cui, se si vuole far terminare le stringhe con un carattere di *newline*, è necessario aggiungerlo in modo esplicito:

```
>>> f = open('myfile', 'w+')
>>> f.writelines(['prima linea\n', 'seconda linea\n'])
>>> f.seek(0)
0
>>> f.read()
'prima linea\nseconda linea\n'
>>> f.tell()
26
>>> d = dict(nome='Max', cognome='Born')
>>> f.writelines(d)
>>> f.seek(26)
26
>>> f.read()
'nomecognome'
```

Il metodo `f.readline()` restituisce una linea del file, lasciando il carattere di *newline* alla fine della stringa. Quando viene raggiunta la fine del file, `f.readline()` restituisce una stringa vuota:

```
>>> f = open('myfile')
>>> f.read()
'prima linea\nseconda linea\nnomecognome'
>>> f.seek(0)
0
>>> f.readline()
'prima linea\n'
>>> f.readline()
'seconda linea\n'
>>> f.readline()
'nomecognome'
>>> f.readline()
''
```

Il metodo `f.readlines()` restituisce una lista contenente tutte le linee del file:

```
>>> open('myfile').readlines()
['prima linea\n', 'seconda linea\n', 'nomecognome']
```

È possibile passare a `f.readlines()` un parametro opzionale, utilizzato per indicare il massimo numero di byte da leggere. La linea viene comunque completata:

```
>>> open('myfile').readlines(5)
['prima linea\n']
>>> open('myfile').readlines(15)
['prima linea\n', 'seconda linea\n']
>>> open('myfile').readlines(30)
['prima linea\n', 'seconda linea\n', 'nomecognome']
```

Iterare sulle linee di un file

I file objects sono degli iteratori, e iterano sulle linee del file:

```
>>> f = open('myfile', 'w+')
>>> hasattr(f, '__iter__'), hasattr(f, '__next__')
(True, True)
>>> f.writelines(['prima\n', 'seconda\n', 'terza\n'])
>>> f.seek(0)
0
>>> next(f), next(f), next(f)
('prima\n', 'seconda\n', 'terza\n')
>>> a, *b = open('myfile')
>>> a, b
('prima\n', ['seconda\n', 'terza\n'])
```

Per leggere le linee di un file una alla volta, si deve iterare sul file object e non sulla lista delle sue linee:

```
>>> for line in open('myfile'):
...     print(line, end='')
...
prima
seconda
terza
```

Il motivo è chiaro se andiamo a vedere cosa accade in entrambi i casi, utilizzando ancora una volta il modulo `dis`:

```
>>> f = open('myfile')
>>> def pythonic_way():
...     for line in f:
...         pass
...
>>> def bad_way():
...     for line in f.readlines():
...         pass
...
```

```

>>> import dis
>>> dis.dis(pythonic_way)
2 0 SETUP_LOOP 14 (to 17)
3 LOAD_GLOBAL 0 (f)
6 GET_ITER
>> 7 FOR_ITER 6 (to 16)
10 STORE_FAST 0 (line)
3 13 JUMP_ABSOLUTE 7
>> 16 POP_BLOCK
>> 17 LOAD_CONST 0 (None)
20 RETURN_VALUE
>>> dis.dis(bad_way)
2 0 SETUP_LOOP 20 (to 23)
3 LOAD_GLOBAL 0 (f)
6 LOAD_ATTR 1 (readlines)
9 CALL_FUNCTION 0 (0 positional, 0 keyword pair)
12 GET_ITER
>> 13 FOR_ITER 6 (to 22)
16 STORE_FAST 0 (line)
3 19 JUMP_ABSOLUTE 13
>> 22 POP_BLOCK
>> 23 LOAD_CONST 0 (None)
26 RETURN_VALUE

```

La funzione `bad_way()` ha due istruzioni bytecode in più: una per caricare l'attributo `f.readlines` (`LOAD_ATTR readlines`) e l'altra per chiamarlo (`CALL_FUNCTION`). L'iterazione sul file object risulta, infatti, più veloce:

```

$ python -m timeit "for line in open('myfile'): pass"
10000 loops, best of 3: 43.8 usec per loop
$ python -m timeit "for line in open('myfile').readlines(): pass"
10000 loops, best of 3: 48 usec per loop

```

Quando si ha terminato di utilizzare il file, si può chiamare il metodo `f.close()`, il quale libera le risorse di sistema allocate dalla funzione `open()`, e, come abbiamo visto, comporta lo svuotamento del buffer di uscita. Una volta che il file viene chiuso, non può più essere letto:

```

>>> f.close()
>>> f.closed
True
>>> f.read()
Traceback (most recent call last):
...
ValueError: I/O operation on closed file.

```


Quando si apre un file, è buona pratica utilizzare l'istruzione `with`. Il vantaggio di questo approccio consiste nel fatto che il file viene chiuso correttamente quando si esce dal blocco, anche se viene lanciata una eccezione:

```
>>> with open('myfile') as f:
...     data = f.read()
...     raise Exception('Errore durante la lettura del file') # Solleviamo una eccezione
...
Traceback (most recent call last):
...
Exception: Errore durante la lettura del file
>>> f.closed
True
```

Per brevità, nel prosieguo del capitolo, quando apriremo un file non utilizzeremo l'istruzione `with`. Il discorso, comunque, non finisce qui: ne parleremo in dettaglio nel [Capitolo 5](#).

I parametri della funzione `open()`

La forma generale della funzione `open()` è la seguente:

```
open(
    file,
    mode='r',
    buffering=-1,
    encoding=None,
    errors=None,
    newline=None,
    closefd=True,
    opener=None
) -> file object
```

Il parametro file

Con la maggior parte dei sistemi operativi moderni è possibile utilizzare caratteri Unicode arbitrari nel nome dei file. Usualmente i sistemi operativi convertono la stringa Unicode utilizzando la propria codifica, che possiamo scoprire chiamando `sys.getfilesystemencoding()`:

```
>>> import sys
>>> sys.platform
'linux'
>>> sys.getfilesystemencoding() # Codifica di default del sistema
'utf-8'
```

La stringa passata come primo argomento, che identifica il nome del file, viene automaticamente codificata da Python utilizzando la codifica di default del sistema operativo:

```
>>> filename = 'myfile\u2160' # L'ultimo carattere è il numero romano 1
>>> print(filename)
myfile I
>>> f = open(filename, 'w')
>>> f.write('Python 3')
8
>>> f.name
'myfile I '
```

La funzione `open()` consente anche di creare un file object a partire da un file descriptor. Per fare ciò si utilizza come primo argomento il *file descriptor*:

```
>>> f1 = open('myfile', 'w+')
>>> f1.fileno() # Restituisce il file descriptor
>>> 3
>>> f2 = open(f1.fileno())
```

Approfondiremo la discussione sui file objects e i file descriptor nella sezione *Closefd*.

Mode

Di seguito sono riportati tutti i possibili valori del parametro `mode`:

- `r`: *reading* (default), apre il file in lettura;
- `w`: *writing*, apre il file in scrittura. Se il file esiste, viene sovrascritto, altrimenti viene creato;
- `x`: apre il file in scrittura. Se il file non esiste, viene creato, altrimenti viene lanciata una eccezione di tipo `FileExistsError`;
- `a`: *appending*, apre il file in scrittura. Se il file esiste, allora la scrittura viene accodata alla fine del file;
- `b`: *binary mode*, il file viene aperto in modalità binaria, per cui l'output sarà una stringa di byte, e anche per l'input ci si aspettano byte. Deve essere usata assieme alle altre modalità di apertura: `rb`, `wb`, `xb` ecc.;
- `t`: *text mode* (default), il file viene aperto in modalità testo, quindi l'output sarà una stringa di testo e anche l'input dovrà esserlo. Deve essere usata assieme alle altre modalità di apertura;
- `+`: le modalità `r+`, `w+`, `x+` e `a+` sono equivalenti alle modalità `r`, `w`, `x` e `a`, ma il file object risulterà sia leggibile sia scrivibile.

Abbiamo già visto nei precedenti esempi tutte le modalità a eccezione di `a` e `x`. La modalità `a` consente di scrivere a partire dalla fine del file:

```
>>> open('myfile', 'w').writelines(['prima\n', 'seconda\n', 'terza\n'])
>>> open('myfile', 'a').write('quarta\n')
7
>>> open('myfile').read()
'prima\nseconda\nterza\nquarta\n'
```

La modalità `x`, introdotta a partire da Python 3.3, consente di evitare già ad alto livello alcune tecniche di *symlink attacks*, come, ad esempio, il seguente scenario:

- un programma *P*, quando viene eseguito, crea un file temporaneo `_nome_programma_tmp_`, senza verificare preventivamente che questo esista;
- un *attacker* riesce a creare un link simbolico di nome `_nome_programma_tmp_` verso un file importante di nome `file_importante`.

A questo punto, quando il programma *P* scrive su `_nome_programma_tmp_`, in realtà sovrascrive `file_importante`. Supponiamo di avere un file importante, di nome `file_importante`:

```
$ echo "Contenuto molto importante..." > file_importante
```

Un *attacker* crea un link simbolico a `file_importante`, sapendo che ci sarà un programma che creerà un file avente il nome dato al link simbolico, senza verificarne preventivamente l'esistenza:

```
$ ln -s $PWD/file_importante $PWD/_nome_programma_tmp_
$ ls -l
file_importante
_nome_programma_tmp_ -> /home/marco/temp/file_importante
$ head file_importante
Contenuto molto importante...
```

La nostra applicazione vorrebbe creare `_nome_programma_tmp_`, ma il file esiste ed è un link simbolico che punta a `file_importante`:

```
$ python -c "open('_nome_programma_tmp_', 'w').write('ops...')"
```

Il file importante è stato sovrascritto:

```
$ head file_importante
ops...
```

Per scongiurare questo tipo di attacchi, occorre accertarsi che il file che si vuole creare non esista. Questo, sino a Python 3.2, lo si faceva creando il file a basso livello, utilizzando la funzione `os.open()` con il flag `os.O_EXCL`:

```
>>> fd = os.open('_nome_programma_tmp_', os.O_WRONLY | os.O_EXCL | os.O_CREAT)
Traceback (most recent call last):
...
FileExistsError: [Errno 17] File exists: '_nome_programma_tmp_'
```

Adesso è possibile ottenere lo stesso risultato anche ad alto livello, mediante la funzione built-in `open()` con la modalità di apertura `x`:

```
>>> file = open('_nome_programma_tmp_', 'x')
Traceback (most recent call last):
...
FileExistsError: [Errno 17] File exists: '_nome_programma_tmp_'
```

Sempre con Python 3.3 è stato introdotto il parametro `opener`, che, come vedremo, consente anche di estendere l'utilizzo della funzione built-in `open()` a tutti i casi per i quali era prima necessario lavorare a basso livello tramite i file descriptor.

Buffering

Il parametro opzionale `buffering` è un numero intero utilizzato per impostare la strategia di buffering. A prescindere dal valore assegnato al parametro `buffering`, il contenuto del buffer viene sempre trasferito sul file sottostante quando si fa una chiamata a uno dei seguenti metodi del file object: `flush()`, `seek()`, `tell()` o `close()`.

Il valore `0` disabilita il buffering:

```
>>> f = open('myfile', 'wb', buffering=0)
>>> f.write(b'Senza buffering') # Scrive immediatamente sul file system
15
>>> open('myfile').read() # Posso leggere il contenuto del file
'Senza buffering'
```

Può essere utilizzato solo in modalità binaria:

```
>>> f = open('myfile', 'w', buffering=0)
Traceback (most recent call last):
...
ValueError: can't have unbuffered text I/O
```

Il valore 1, quando utilizzato in modalità testo, imposta il buffering di linea, ovvero il buffer viene svuotato automaticamente a ogni newline:

```
>>> f = open('myfile', 'w', buffering=1)
>>> f.line_buffering
True
>>> f.write('Domani ')
7
>>> open('myfile').read()
"
>>> f.write('vado al mare\n')
13
>>> open('myfile').read()
'Domani vado al mare\n'
```

Un valore del buffering maggiore di 1 definisce, invece, la *profondità del buffer*. In questo caso la politica di buffering è legata, oltre che al parametro buffering, anche a un'altra variabile, detta `CHUNK_SIZE`. Infatti, quando si scrive sul file object, i byte non vengono trasferiti immediatamente sul buffer, ma vengono messi in coda in attesa che il loro numero superi `CHUNK_SIZE`. Questa strategia di buffering è spiegata in dettaglio nell'*Appendice C*.

Encoding e errors

Il parametro `encoding` è utilizzato per definire la codifica da utilizzare per leggere il contenuto del file. Se non viene specificata alcuna codifica, verrà utilizzata quella impostata nel sistema operativo:

```
>>> import sys
>>> sys.platform, sys.getdefaultencoding()
('linux', 'utf-8')
>>> f = open('myfile') # Viene utilizzata la codifica del sistema operativo
>>> f.encoding
'UTF-8'
>>> f = open('myfile', encoding='latin1')
>>> f.encoding
'latin1'
```

Come sappiamo, la codifica determina il modo in cui le stringhe di testo verranno convertite in stringhe di byte:

```
>>> f1 = open('myfile1', 'w')
>>> f2 = open('myfile2', 'w', encoding='latin1')
>>> s = 'Questo è il contenuto'
>>> f1.write(s)
21
>>> f2.write(s)
21
```

```
>>> f1.close()
>>> f2.close()
>>> open('myfile1', 'rb').read()
b'Questo \xc3\xa8 il contenuto'
>>> open('myfile2', 'rb').read()
b'Questo \xe8 il contenuto'
```

La codifica determina anche il modo in cui le stringhe di byte verranno convertite in stringhe di testo:

```
>>> open('myfile1').read()
'Questo è il contenuto'
>>> open('myfile2', encoding='latin1').read()
'Questo è il contenuto'
>>> open('myfile2').read() # Decodifica in UTF-8, ma il testo è codificato in latin1
Traceback (most recent call last):
...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe8 in position 7: invalid continuation byte
```

L'argomento `encoding` può essere passato solo in modalità testo:

```
>>> f = open('myfile', 'w', encoding='latin1')
>>> f = open('myfile', 'wb', encoding='latin1')
Traceback (most recent call last):
...
ValueError: binary mode doesn't take an encoding argument
```

Il parametro `errors`, utilizzabile solo in modalità testo, ci consente di gestire gli errori. Quando si ha `errors='strict'` viene lanciata, in caso di errore di codifica, una eccezione di tipo `UnicodeDecodeError`, come nell'esempio visto sopra (il valore di default `None` ha, infatti, lo stesso effetto di `strict`). Quando `errors='ignore'`, gli errori vengono ignorati, mentre quando `errors='replace'` il carattere che non si riesce a decodificare viene sostituito:

```
>>> open('myfile2', errors='ignore').read() # Converte in UTF-8 ma ignora l'errore
'Questo il contenuto'
>>> open('myfile2', errors='replace').read()
'Questo ? il contenuto'
```

NOTA

Si osservi che il metodo da utilizzare per decodificare il contenuto di un file di testo è quello appena illustrato, dove il tipo di codifica viene assegnato al parametro `encoding` nella chiamata a `open()`. Non dobbiamo, infatti, leggere il contenuto del file come stringa di byte, disinteressandoci, quindi, del tipo di codifica, per poi decodificare successivamente la stringa con il metodo

`bytes.decode()`. A causa della natura multi-byte dell'Unicode, infatti, questo approccio è da evitare perché si è costretti a leggere l'intero contenuto, altrimenti si rischia di spezzare un gruppo di byte che formano un carattere:

```
>>> s = open('myfile1', 'rb').read(8)
>>> s.decode(encoding='utf-8')
Traceback (most recent call last):
...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 7: unexpected end of data
>>> s = open('myfile1').read(8) # Metodo corretto
>>> s
'Questo è'
```

Se si utilizzano le codifiche UTF-16 e UTF-32, durante la scrittura (modalità testo, ovviamente) il BOM viene automaticamente inserito come primo carattere, mentre durante la lettura (modalità testo) viene ignorato:

```
>>> open('myfile', 'w', encoding='utf-16').write('Python 3')
8
>>> open('myfile', 'rb').read()
b'\xff\xfeP\x00y\x00t\x00h\x00o\x00n\x00 \x003\x00'
>>> open('myfile', 'rb').read(2) # Il BOM è presente
b'\xff\xfe'
>>> open('myfile', encoding='utf-16').read() # Il BOM viene ignorato
'Python 3'
```

Newline

Questo parametro serve per stabilire la politica da utilizzare per il controllo dell'*universal newline* (vedi la PEP-0278). Può assumere i valori `None`, `"`, `'\n'`, `'\r'` e `'\r\n'` e può essere utilizzato solo in modalità testuale:

```
>>> f = open('myfile', 'rb', newline='\r\n')
Traceback (most recent call last):
...
ValueError: binary mode doesn't take a newline argument
```

Vediamo prima come influisce questo parametro quando il file viene letto. Consideriamo, a tale scopo, il seguente file:

```
>>> open('myfile', 'rb').read()
b'prima linea\rseconda linea\rterza linea\nquarta linea\r\n'
```

Se `newline="`, le linee vengono terminate in corrispondenza dei terminatori di linea (`'\n'`, `'\r'` e `'\r\n'`) e questi vengono restituiti inalterati:

```
>>> [line for line in open('myfile', newline='')]
['prima linea\r', 'seconda linea\r', 'terza linea\n', 'quarta linea\r\n']
```

Se `newline=None` (default), tutte le sequenze `'\n'`, `'\r'` o `'\r\n'` vengono ancora considerate terminatori di linea, ma sono convertite in `'\n'` prima di essere restituite:

```
>>> [line for line in open('myfile')]
['prima linea\n', 'seconda linea\n', 'terza linea\n', 'quarta linea\n']
```

Se `newline` assume uno qualsiasi dei restanti valori, allora solo questo valore è considerato terminatore di linea e tutti i caratteri vengono restituiti inalterati:

```
>>> [line for line in open('myfile')]
['prima linea\n', 'seconda linea\n', 'terza linea\n', 'quarta linea\n']
>>> [line for line in open('myfile', newline='\n')]
['prima linea\rseconda linea\rterza linea\n', 'quarta linea\r\n']
>>> [line for line in open('myfile', newline='\r')]
['prima linea\r', 'seconda linea\r', 'terza linea\nquarta linea\r', '\n']
>>> [line for line in open('myfile', newline='\r\n')]
['prima linea\rseconda linea\rterza linea\nquarta linea\r\n']
```

Se, invece, consideriamo le scritture su file, allora quando `newline=None` ogni `'\n'` viene sostituito con il separatore di linea di sistema:

```
>>> import os, sys
>>> sys.platform
'win32'
>>> os.linesep # Separatore di linea di sistema
'\r\n'
>>> open('myfile', 'w').write('prima linea\n')
12
>>> open('myfile', 'rb').read()
b'prima linea\r\n'
>>> open('myfile').read()
'prima linea\n'
```

Se, invece, `newline=''` o `newline='\n'`, non avviene nessuna alterazione:

```
>>> f = open('myfile', 'w', newline='')
>>> f.writelines(['prima\n', 'seconda\r\n', 'terza\r'])
>>> f.close()
>>> open('myfile', 'rb').read()
b'prima\nseconda\r\nterza\r'
```

Se, invece, `newline` prende uno dei restanti valori, allora ogni carattere di `'\n'` viene convertito nella stringa assegnata a `newline`:


```
>>> f = open('myfile', 'w', newline='\r')
>>> f.write('prima linea\nseconda linea\r\n')
27
>>> f.close()
>>> open('myfile', 'rb').read()
b'prima linea\rseconda linea\r'
```

Closefd

Prima di parlare del significato del parametro `closefd`, chiariamo il concetto di *file descriptor*. I file descriptor di un processo sono dei numeri interi associati ai file aperti dal processo. Secondo lo standard POSIX, ogni processo dovrebbe associare i file descriptor 0, 1 e 2 rispettivamente allo standard input, output ed error. Il metodo `fileno()` di un file object restituisce il file descriptor associato al file:

```
>>> import sys
>>> sys.stdin.fileno(), sys.stdout.fileno(), sys.stderr.fileno()
(0, 1, 2)
>>> open('myfile').fileno()
3
```

Un file object `f2` creato sulla base del file descriptor di `f1` ha lo stesso file descriptor di quest'ultimo e i due file object condividono la medesima risorsa:

```
>>> f1 = open('myfile', 'w')
>>> f2 = open(f1.fileno())
>>> f1.fileno(), f2.fileno()
(3, 3)
>>> f1.name, f2.name
('myfile', 3)
>>> f1.write('Python 3')
8
>>> f1.flush()
>>> f2.seek(0)
0
>>> f2.read()
'Python 3'
```

Il parametro `closefd` non ha alcuna influenza sui file object creati utilizzando il nome del file. Per i file creati sulla base di un file descriptor, invece, quando `closefd=True` (valore di default) la chiusura di un file object comporta la chiusura del file descriptor, rendendo inutilizzabili tutti gli altri file object che condividevano il file descriptor:

```
>>> f1 = open('myfile', 'w')
>>> f2 = open(f1.fileno())
>>> import subprocess, os
```

```
>>> subprocess.call('ls -l /proc/{}/fd/'.format(os.getpid()), shell=True)
total 0
lrwx----- 1 marco marco 64 2012-08-23 19:04 0 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 19:04 1 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 19:04 2 -> /dev/pts/4
lr-x----- 1 marco marco 64 2012-08-23 19:04 3 -> /home/marco/test/myfile
0
>>> f2.close()
>>> subprocess.call('ls -l /proc/{}/fd/'.format(os.getpid()), shell=True)
total 0
lrwx----- 1 marco marco 64 2012-08-23 19:04 0 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 19:04 1 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 19:04 2 -> /dev/pts/4
0
>>> f1.close() # Il file descriptor 3 non esiste più...
Traceback (most recent call last):
...
OSError: [Errno 9] Bad file descriptor
```

NOTA

Il modulo `subprocess` della libreria standard consente di creare dei processi, connettersi alla loro pipe di input, output ed error e ottenere il loro stato di uscita. In particolare, la funzione `subprocess.call()` crea un nuovo processo e restituisce il suo stato di uscita:

```
>>> subprocess.call('pwd')
/home/marco
0
```

Come possiamo vedere da questo esempio, `subprocess.call('pwd')` ha chiamato il comando `pwd`; questo ha stampato la directory di lavoro e ha restituito `0`. Se dobbiamo passare degli argomenti al comando, allora abbiamo due alternative. La prima consiste nel passare a `subprocess.call()` una lista contenente come elementi il comando e i suoi argomenti:

```
>>> subprocess.call(['ls', '/home/marco/myvenv'])
bin include lib man pyvenv.cfg share
0
```

La seconda, invece, fa uso dell'argomento opzionale `shell`, che per default è impostato a `False`. Se poniamo `shell=True`, allora i comandi verranno eseguiti nella shell e in questo caso possiamo passare anche una stringa contenente il nome del comando e i suoi switch e argomenti:

```
>>> subprocess.call('ls /home/marco/myvenv', shell=True)
bin include lib man pyvenv.cfg share
```

In quest'ultimo caso, il comando viene eseguito nella shell, per cui possiamo usare variabili d'ambiente, wildcard e quant'altro:

```
>>> subprocess.call('ls /home/marco/myenv/*.*cfg', shell=True)
/home/marco/myenv/pyvenv.cfg
0
>>> subprocess.call('echo $HOME', shell=True)
/home/marco
0
```

Si osservi che eseguire i comandi direttamente nella shell rende il programma vulnerabile ad attacchi di *shell injection*:

```
>>> path = input('Inserisci il percorso della directory: ')
Inserisci il percorso della directory: ATTENZIONE;rm * -rf
>>> subprocess.call('ls ' + path, shell=True) # Non eseguirlo!
```

Per maggiori informazioni sul modulo `subprocess` possiamo consultare la documentazione ufficiale, alla pagina <http://docs.python.org/3/library/subprocess.html>.

Quando, invece, si lavora ad alto livello, utilizzando il nome del file, vengono creati file descriptor diversi nonostante la risorsa condivisa sia la medesima:

```
>>> f1 = open('myfile')
>>> f2 = open('myfile')
>>> f1.fileno(), f2.fileno()
(3, 4)
>>> import subprocess, os
>>> subprocess.call('ls -l /proc/{}/fd/'.format(os.getpid()), shell=True)
total 0
lrwx----- 1 marco marco 64 2012-08-23 20:53 0 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 20:53 1 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 20:53 2 -> /dev/pts/4
lr-x----- 1 marco marco 64 2012-08-23 20:53 3 -> /home/marco/test/myfile
lr-x----- 1 marco marco 64 2012-08-23 20:53 4 -> /home/marco/test/myfile
0
>>> f1.close()
>>> subprocess.call('ls -l /proc/{}/fd/'.format(os.getpid()), shell=True)
total 0
lrwx----- 1 marco marco 64 2012-08-23 20:53 0 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 20:53 1 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 20:53 2 -> /dev/pts/4
lr-x----- 1 marco marco 64 2012-08-23 20:53 4 -> /home/marco/test/myfile
0
>>> f2.close()
```

```
>>> subprocess.call('ls -l /proc/{}/fd/'.format(os.getpid()), shell=True)
total 0
lrwx----- 1 marco marco 64 2012-08-23 20:53 0 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 20:53 1 -> /dev/pts/4
lrwx----- 1 marco marco 64 2012-08-23 20:53 2 -> /dev/pts/4
0
```

Opener

Il parametro `opener`, introdotto a partire da Python 3.3, consente di estendere l'utilizzo della funzione built-in `open()` anche a quei casi per i quali prima era necessario lavorare direttamente a basso livello. Ad esempio, supponiamo di voler creare un file (solo se non esiste già) aperto in lettura e scrittura e di volergli assegnare i permessi `744`. Prima dell'introduzione di questo nuovo parametro, sarebbe stato necessario utilizzare direttamente il modulo `os`:

```
>>> import os
>>> fd = os.open('myscript.bash', os.O_CREAT | os.O_RDWR | os.O_EXCL, 0o744)
>>> # Facciamo una verifica
... import stat
>>> mode = os.stat('myscript.bash').st_mode
>>> stat.filemode(mode) # Introdotto con Python 3.3
'-rwxr--r--'
```

Adesso è invece possibile fare la stessa cosa direttamente con la funzione built-in `open()`:

```
>>> import os, stat
>>> open('foo.bash', 'x', opener=lambda file, flags: os.open(file, flags, 0o744))
<_io.TextIOWrapper name='foo.bash' mode='x' encoding='UTF-8'>
>>> mode = os.stat('foo.bash').st_mode
>>> stat.filemode(mode)
'-rwxr--r--'
```

Il parametro `opener` deve essere `None` (default) oppure un oggetto *callable* e, in quest'ultimo caso, come abbiamo appena visto, il file descriptor da associare al file object viene ottenuto chiamando `opener(file, flags)`.

Esercizio conclusivo

In questo esercizio conclusivo vedremo un programma che fa uso dei generatori. Il codice è il seguente:

```
$ cat finder.py
#!/usr/bin/env python
"""Cerca dei file con un dato nome, e anche del testo al loro interno."""

import argparse
import fnmatch
import os
import re

parser = argparse.ArgumentParser(description='Cerca sia file che testo')
parser.add_argument('-f', '--file', type=str, required=True, help='Nome del file')
parser.add_argument('-d', '--dir', default=os.curdir, help='Directory di partenza')
parser.add_argument('-p', '--pattern', default="", help='Pattern da cercare')
parser.add_argument('-r', '--recursive', action='store_true', help='Ricorsiva')
args = parser.parse_args()

def file_finder(pattern: str, top_dir: str, recursive: bool=False):
    for path, dirs, files in os.walk(top_dir):
        if not recursive:
            dirs.clear() # Svuota la lista delle sotto-directory di `top_dir`
        for name in fnmatch.filter(files, pattern):
            yield os.path.join(path, name)

def file_inspector(file_name: str, pattern: str):
    for line in open(file_name):
        if re.search(pattern, line):
            yield line

for file in file_finder(args.file, args.dir, args.recursive):
    if args.pattern:
        for line in file_inspector(file, args.pattern):
            print(file, line, sep=' -> ', end='')
    else:
        print(file)
```

Questo script consente di compiere sia una ricerca di file, sia una ricerca di testo all'interno dei file.

Come al solito, nelle prossime sezioni vedremo in dettaglio il significato del codice. Per il momento proviamo a leggere ancora lo script, sforzandoci di capirne il significato in modo autonomo.

Esecuzione dello script

Iniziamo subito con un esempio di utilizzo. Consideriamo, a tale scopo, la seguente directory *mydir*:

```
$ ls /home/marco/temp/mydir/  
dirA dirB myfile.txt myfoo.txt
```

Se eseguiamo lo script nel modo seguente:

```
$ python finder.py -f "*foo*" -d /home/marco/temp/mydir/  
/home/marco/temp/mydir/myfoo.txt
```

allora vengono mostrati a video i file il cui nome contiene il testo `foo`. Se allo script viene passata da linea di comando l'opzione `-r`, allora la ricerca avviene in modo ricorsivo:

```
$ python finder.py -f "*foo*" -d /home/marco/temp/mydir/ -r  
/home/marco/temp/mydir/myfoo.txt  
/home/marco/temp/mydir/dirA/afoofile.txt  
/home/marco/temp/mydir/dirA/foo  
/home/marco/temp/mydir/dirB/bfoofile.txt
```

Oltre alle opzioni `-f`, `-d` e `-r`, è possibile utilizzare anche l'opzione `-p`:

```
$ python finder.py -f "*foo*" -d /home/marco/temp/mydir/ -p lin -r  
/home/marco/temp/mydir/dirA/afoofile.txt -> prima linea  
/home/marco/temp/mydir/dirA/afoofile.txt -> seconda linea  
/home/marco/temp/mydir/dirB/bfoofile.txt -> prima linea  
/home/marco/temp/mydir/dirB/bfoofile.txt -> seconda linea
```

Come possiamo vedere, lo script ha stampato quattro linee, in ciascuna delle quali è presente sia il nome di un file sia la linea del file nella quale è presente il testo `lin` separati da una freccia. Sostanzialmente, utilizzando l'opzione `-p` lo script si comporta come il comando Unix `grep`:

```
$ grep lin /home/marco/temp/mydir/ -R  
/home/marco/temp/mydir/dirA/afoofile.txt:prima linea  
/home/marco/temp/mydir/dirA/afoofile.txt:seconda linea  
/home/marco/temp/mydir/dirB/bfoofile.txt:prima linea  
/home/marco/temp/mydir/dirB/bfoofile.txt:seconda linea
```

Nell'esempio precedente il testo `lin` è stato cercato solo nei file contenenti il testo `foo` nel loro nome. È possibile anche cercare il testo `lin` in tutti i file, a prescindere, quindi, dal loro nome:

```
$ python finder.py -f "*" -d /home/marco/temp/mydir/ -p line -r
/home/marco/temp/mydir/myfile.txt -> unica linea
/home/marco/temp/mydir/dirA/afoofile.txt -> prima linea
/home/marco/temp/mydir/dirA/afoofile.txt -> seconda linea
/home/marco/temp/mydir/dirB/bfoofile.txt -> prima linea
/home/marco/temp/mydir/dirB/bfoofile.txt -> seconda linea
```

Il programma può fare molto di più, come vedremo nelle prossime sezioni.

Ricerca dei file

La ricerca dei file viene effettuata grazie alla funzione generatore `file_finder()`:

```
def file_finder(pattern: str, top_dir: str, recursive: bool=False):
    for path, dirs, files in os.walk(top_dir):
        if not recursive:
            dirs.clear() # Svuota la lista delle sotto-directory di `top_dir`
        for name in fnmatch.filter(files, pattern):
            yield os.path.join(path, name)
```

Nonostante sia composta solamente da sei linee di codice, non è per nulla banale. Iniziamo con il vedere la funzione `os.walk()`.

Percorrere le directory

La funzione `os.walk()` ci consente di percorrere un albero di directory. Prende un argomento posizionale obbligatorio, che chiamiamo *top-directory*, e restituisce un generatore:

```
>>> import os
>>> g = os.walk('/boot/efi')
>>> g
<generator object walk at 0x7faa1dd14500>
```

Il generatore a ogni richiesta produce una tupla contenente il percorso della directory ispezionata, la lista delle directory contenute all'interno della directory ispezionata e i file in essa presenti. Nel nostro caso, ad esempio, alla prima iterazione il generatore produce la seguente tupla:

```
>>> next(g)
('/boot/efi', ['EFI', 'Temp', 'linuxmint', 'Microsoft', 'Boot'], [])
```

Il primo elemento della tupla è la stringa `'/boot/efi'`, che rappresenta il percorso della top-directory. Il secondo elemento della tupla è la lista delle directory presenti in `/boot/efi/`, ovvero `['EFI', 'Temp', 'linuxmint', 'Microsoft', 'Boot']`. Il terzo elemento è una lista vuota, perché nella directory `/boot/efi/` non è presente alcun file.

Alla seconda iterazione il generatore produce la seguente tupla:

```
>>> next(g)
('/boot/efi/EFI', ['Microsoft', 'Boot', 'ASUS', 'linuxmint', 'grub'], [])
```

Come possiamo vedere, la directory ispezionata è */boot/efi/EFI*.

Alla terza iterazione si va ancora più in profondità, ispezionando la directory */boot/efi/EFI/Microsoft*:

```
>>> next(g)
('/boot/efi/EFI/Microsoft', ['Boot'], [])
```

Vediamo un esempio di iterazione completa. Consideriamo, a tale scopo, la seguente gerarchia di directory:

```
$ tree /home/marco/temp/mydir/
/home/marco/temp/mydir/
├── dirA
│   ├── afoofile.txt
│   └── foo
├── dirB
│   └── bfoofile.txt
├── myfile.txt
└── myfoo.txt
2 directories, 5 files
```

Ecco il risultato che si ottiene iterando con `os.walk()`:

```
>>> import os
>>> for path, dirs, files in os.walk('/home/marco/temp/mydir'):
...     print(path, dirs, files)
...
/home/marco/temp/mydir ['dirA', 'dirB'] ['myfoo.txt', 'myfile.txt']
/home/marco/temp/mydir/dirA [] ['afoofile.txt', 'foo']
/home/marco/temp/mydir/dirB [] ['bfoofile.txt']
```

La lista `dirs` delle directory viene utilizzata da `os.walk()` come elenco delle directory da ispezionare, per cui, se rimuoviamo una directory da tale lista, `os.walk()` non la ispeziona:

```
>>> for path, dirs, files in os.walk('/home/marco/temp/mydir'):
...     print(path, dirs, files)
...     if 'dirA' in dirs:
...         dirs.remove('dirA')
...
/home/marco/temp/mydir ['dirA', 'dirB'] ['myfoo.txt', 'myfile.txt']
```



```
/home/marco/temp/mydir/dirB [] ['bfoofile.txt']
```

Quindi, se alla prima iterazione svuotiamo la lista, verrà ispezionata solamente la top-directory:

```
>>> for path, dirs, files in os.walk('/home/marco/temp/mydir'):
...     print(path, dirs, files)
...     dirs.clear()
...
/home/marco/temp/mydir ['dirA', 'dirB'] ['myfoo.txt', 'myfile.txt']
```

Questo è il motivo per cui nella funzione `file_finder()` sono presenti le seguenti linee di codice:

```
if not recursive:
    dirs.clear() # Svuota la lista delle sotto-directory di `top_dir`
```

Se guardiamo la signature della funzione `file_finder()` vedremo che per default si ha `recursive=True`, il che significa che la ricerca viene fatta in modo ricorsivo. Se, invece, viene passato `recursive=False`, allora la ricerca si ferma alla *top-directory*.

Il supporto per le wildcard

Il modulo `fnmatch` della libreria standard fornisce il supporto per i *caratteri jolly* utilizzati nelle shell Unix. Nella lingua inglese i caratteri jolly vengono usualmente chiamati *wildcard characters*. Per capire ciò di cui stiamo parlando, iniziamo con il vedere il significato della wildcard più utilizzata nelle shell Unix: l'*asterisco*. Questo rappresenta una qualsiasi sequenza di caratteri (con la sola eccezione dei caratteri di significato speciale come `.` e `/`), inclusa la sequenza vuota. Ad esempio, il pattern `*.py` combacia con tutte le stringhe che terminano con `.py`, compresa la stringa `.py` stessa. Il pattern `f*.py` combacia con tutte le stringhe che iniziano con il carattere `f` e terminano con `.py`, come, ad esempio, `foo.py`, `file.py` ecc. Consideriamo, ad esempio, il comando `ls` dei sistemi Unix-like:

```
$ ls
fileA.py fileA.txt fileB.py fileB.txt foo.py myfile.py README setup.py
```

Se vogliamo che vengano mostrati solamente i nomi dei file e delle directory che terminano con `.py`, utilizziamo il carattere asterisco nel seguente modo:

```
$ ls *.py
```

```
fileA.py fileB.py foo.py myfile.py setup.py
```

Se, invece, vogliamo vedere i nomi dei file e delle directory che iniziano con il carattere `f` e terminano con `.py`, facciamo come mostrato di seguito:

```
$ ls f*.py
fileA.py fileB.py foo.py
```

Ecco, invece, i nomi dei file che contengono la sequenza di caratteri `le`:

```
$ ls *le*
fileA.py fileA.txt fileB.py fileB.txt myfile.py
```

Come probabilmente abbiamo intuito, il comando `ls *` si comporta in modo equivalente al comando `ls` utilizzato senza parametri, poiché visualizza tutto il contenuto della directory:

```
$ ls
fileA.py fileA.txt fileB.py fileB.txt foo.py myfile.py README setup.py
$ ls *
fileA.py fileA.txt fileB.py fileB.txt foo.py myfile.py README setup.py
```

Il modulo `fnmatch` ci consente di utilizzare le wildcard delle shell Unix. Ad esempio, la funzione `fnmatch.filter()` prende come primo argomento una sequenza di nomi e come secondo argomento un *pattern*, ovvero una stringa nella quale possono essere presenti anche delle wildcard da interpretare, e restituisce la lista dei nomi che verificano il pattern.

Consideriamo la stessa directory dell'esempio precedente. Come sappiamo, la funzione `os.listdir()` chiamata senza argomenti restituisce una lista dei nomi dei file e delle directory presenti nella directory corrente:

```
>>> import os
>>> os.listdir()
['fileB.py', 'foo.py', 'fileA.py', 'README', 'fileA.txt', 'setup.py', 'fileB.txt', 'myfile.py']
```

Possiamo passare `os.listdir()` come primo argomento della funzione `fnmatch.filter()`, mentre come secondo argomento possiamo passare il pattern che abbiamo utilizzato nei precedenti esempi (gli argomenti del comando `ls`):

```
>>> import fnmatch
>>> fnmatch.filter(os.listdir(), '*.py')
['fileB.py', 'foo.py', 'fileA.py', 'setup.py', 'myfile.py']
>>> fnmatch.filter(os.listdir(), 'f*.py')
['fileB.py', 'foo.py', 'fileA.py']
```

```
>>> fnmatch.filter(os.listdir(), '*le*')
['fileB.py', 'fileA.py', 'fileA.txt', 'fileB.txt', 'myfile.py']
>>> fnmatch.filter(os.listdir(), '*')
['fileB.py', 'foo.py', 'fileA.py', 'README', 'fileA.txt', 'setup.py', 'fileB.txt', 'myfile.py']
```

Un'altra wildcard è il carattere `?`, il quale rappresenta un carattere qualsiasi:

```
>>> os.listdir()
['fileB.py', 'fileAB.txt', 'fileA.py', 'README', 'fileA.txt', 'fileB.txt', 'fileAB.py']
>>> fnmatch.filter(os.listdir(), 'file?.py')
['fileB.py', 'fileA.py']
>>> fnmatch.filter(os.listdir(), 'file*.py')
['fileB.py', 'fileA.py', 'fileAB.py']
>>> fnmatch.filter(os.listdir(), 'file?.*')
['fileB.py', 'fileA.py', 'fileA.txt', 'fileB.txt']
```

Per maggiori informazioni sul modulo `fnmatch` possiamo consultare la documentazione online sul sito ufficiale. A questo punto il codice della funzione `file_finder()` dovrebbe esserci più chiaro. Vediamolo ancor più in dettaglio.

La funzione generatore `file_finder()`

Consideriamo, ancora una volta, la directory vista in precedenza:

```
$ tree /home/marco/temp/mydir/
/home/marco/temp/mydir/
├── dirA
│   ├── afoofile.txt
│   └── foo
├── dirB
│   └── bfoofile.txt
├── myfile.txt
└── myfoo.txt
2 directories, 5 files
```

Chiamiamo la funzione `file_finder()` passandole come primo argomento un pattern, come secondo argomento la stringa `'/home/marco/temp/mydir'` e come terzo argomento `True`, in modo da effettuare una ricerca ricorsiva:

```
>>> def file_finder(pattern: str, top_dir: str, recursive: bool=False):
...     for path, dirs, files in os.walk(top_dir):
...         if not recursive:
...             dirs.clear() # Svuota la lista delle sotto-directory di `top_dir`
...         for name in fnmatch.filter(files, pattern):
...             yield os.path.join(path, name)
... 
```

```
>>> g = file_finder('*foo*', '/home/marco/temp/mydir/', True)
```

La funzione restituisce un generatore:

```
>>> g
<generator object file_finder at 0x7f3a324f85f0>
```

A ogni iterazione il generatore restituisce un file il cui nome verifica il pattern `*foo*`:

```
>>> next(g)
'/home/marco/temp/mydir/myfoo.txt'
>>> next(g)
'/home/marco/temp/mydir/dirA/afoofile.txt'
>>> next(g)
'/home/marco/temp/mydir/dirA/foo'
>>> next(g)
'/home/marco/temp/mydir/dirB/bfoofile.txt'
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

Per default (`recursive=False`) la ricerca è limitata alla sola top-directory:

```
>>> for file in file_finder('*foo*', '/home/marco/temp/mydir/'):
...     print(file)
...
/home/marco/temp/mydir/myfoo.txt
```

Ecco un altro esempio, utilizzando anche la wildcard `?`:

```
>>> for file in file_finder('?foo*', '/home/marco/temp/mydir/', True):
...     print(file)
...
/home/marco/temp/mydir/dirA/afoofile.txt
/home/marco/temp/mydir/dirB/bfoofile.txt
```

La funzione generatore `file_inspector()`

La funzione `file_inspector()` fa uso del modulo `re` della libreria standard, il quale fornisce il supporto per le espressioni regolari.

Espressioni regolari

Le *espressioni regolari* sono sequenze di caratteri che identificano un insieme di stringhe. Se ci è capitato di utilizzare il comando Unix `grep` probabilmente

sappiamo di cosa stiamo parlando. Infatti *grep* significa *general regular expression print* e consente di cercare all'interno del testo le linee che corrispondono a uno o più pattern specificati tramite espressioni regolari o stringhe letterali, producendo un elenco delle linee per le quali è stata trovata corrispondenza:

```
$ grep "inea" /home/marco/temp/mydir/ -R
/home/marco/temp/mydir/dirA/afoofile.txt:prima linea
/home/marco/temp/mydir/dirA/afoofile.txt:seconda linea
/home/marco/temp/mydir/dirB/bfoofile.txt:prima linea
/home/marco/temp/mydir/dirB/bfoofile.txt:seconda linea
/home/marco/temp/mydir/myfile.txt:unica linea
```

In questo caso *grep* ha cercato il testo *inea* nei file presenti nella directory */home/marco/temp/mydir/* (e nelle sotto-directory di questa, visto che lo switch *-R* impone una ricerca ricorsiva) e ha mostrato a video le linee nelle quali tale testo è presente. Le linee, come possiamo vedere, sono precedute dal percorso del file.

Aggiungiamo ora una nuova linea al file *myfile.txt*:

```
$ echo "abbastanza lineare" >> /home/marco/temp/mydir/myfile.txt
$ grep "inea" /home/marco/temp/mydir/ -R
/home/marco/temp/mydir/dirA/afoofile.txt:prima linea
/home/marco/temp/mydir/dirA/afoofile.txt:seconda linea
/home/marco/temp/mydir/dirB/bfoofile.txt:prima linea
/home/marco/temp/mydir/dirB/bfoofile.txt:seconda linea
/home/marco/temp/mydir/myfile.txt:unica linea
/home/marco/temp/mydir/myfile.txt:abbastanza lineare
```

e vediamo un esempio di pattern che utilizza una espressione regolare:

```
$ grep "inea$" /home/marco/temp/mydir/ -R
/home/marco/temp/mydir/dirA/afoofile.txt:prima linea
/home/marco/temp/mydir/dirA/afoofile.txt:seconda linea
/home/marco/temp/mydir/dirB/bfoofile.txt:prima linea
/home/marco/temp/mydir/dirB/bfoofile.txt:seconda linea
/home/marco/temp/mydir/myfile.txt:unica linea
```

Come possiamo vedere, non è stata trovata corrispondenza con la linea *abbastanza lineare* del file *myfile.txt*. Infatti il carattere *\$* identifica la fine di una riga, per cui è stata trovata corrispondenza con il testo *linea* al termine della riga *unica linea*, ma non con la riga *abbastanza lineare*.

Esercitarsi con le espressioni regolari

Lo scopo di questo esercizio conclusivo non è di certo quello di fornire una guida esaustiva alle espressioni regolari, ma piuttosto di capire cosa sono e come utilizzarle. Come al solito, adotteremo un approccio pratico, per cui il punto di partenza consiste nel trovare uno strumento che ci consenta di effettuare delle prove. Possiamo scegliere un tool online, come ad esempio <http://pythex.org>, mostrato in [Figura 3.1](#).



Figura 3.1 - Il sito web <http://pythex.org> permette di esercitarsi con le espressioni regolari.

Vediamo subito come utilizzare questo strumento. Consideriamo, a tale scopo, il seguente testo:

Il Sardinia Radio Telescope (SRT) è un radiotelescopio situato in Sardegna, a circa 35 km a nord di Cagliari. Ha una imponente struttura meccanica, caratterizzata da uno specchio primario di 64 metri di diametro, una altezza di 70 metri e un peso di 3000 tonnellate. È il più moderno radiotelescopio europeo, sia per quanto riguarda l'equipaggiamento elettronico sia per la componentistica meccanica, i quali lo rendono capace di movimenti di precisione pari a 1/10000 di grado. Il software di controllo del SRT è scritto in Python e C++.

Supponiamo di voler cercare all'interno di esso tutte le occorrenze della sequenza di caratteri `on`. Andiamo, quindi, su <http://pythex.org>, scriviamo `on` nell'area di testo *Your regular expression*, mentre nell'area di testo *Your test string* scriviamo il testo sopra citato. Il risultato è mostrato in [Figura 3.2](#).



Figura 3.2 - Esempio di utilizzo dell'interfaccia di <http://pythex.org>.

A questo punto vogliamo raffinare la ricerca, cercando solamente il testo `on` che si trova al termine di una parola. Per fare ciò possiamo utilizzare il carattere speciale `\b`, che indica per l'appunto il bordo di una parola. La nostra espressione regolare è quindi `on\b` e il risultato della sua ricerca all'interno del testo è mostrato in [Figura 3.3](#). Come possiamo osservare, solamente la parola `Python` combacia con l'espressione regolare `on\b`.



Figura 3.3 - Testo che combacia con l'espressione regolare `on\b`, verificato tramite <http://pythex.org>.

La `b` del carattere speciale `\b` significa *boundary* (limite, delimitazione). Un

delimitatore è qualsiasi carattere che non faccia parte dell'alfabeto, come ad esempio un punto esclamativo, una virgola, uno spazio, degli apici, un punto interrogativo ecc. La [Figura 3.4](#) evidenzia quanto abbiamo appena detto.



Figura 3.4 - Testo che combacia con l'espressione regolare `a\b`, verificato tramite <http://pythex.org>.

Vi sono, ovviamente, altri caratteri speciali. Ad esempio, `\d` combacia con una cifra, `\n` con un fine riga, `\s` con uno spazio, `\A` con l'inizio del testo e via dicendo.

Oltre ai caratteri che sono resi speciali quando preceduti da un backslash, vi è un insieme di caratteri che sono di per sé speciali, come ad esempio il carattere `^`, che identifica l'inizio della linea, il carattere `$`, che identifica la fine della linea e tanti altri.

Anche il backslash è un carattere speciale, allo stesso modo di come lo è per le sequenze di escape delle stringhe, delle quali abbiamo parlato nel [Capitolo 2](#). Consente, infatti, di rappresentare il backslash stesso, per cui l'espressione regolare `\\b` combacia con la sequenza dei due caratteri `\b`. Inoltre il backslash consente di rappresentare i caratteri che di per sé sono speciali, per cui, ad esempio, `\$` combacia con il carattere `$`.

Una nota particolare merita il carattere speciale asterisco (`*`), poiché nelle espressioni regolari ha un significato diverso da quello che ha nelle wildcard. Nelle espressioni regolari, infatti, identifica zero o più occorrenze dell'espressione che lo precede. Ad esempio, il carattere speciale `\d` identifica una cifra, per cui l'espressione regolare `\d*` rappresenta zero o più occorrenze di una cifra, e quindi `Python\d*` combacia con `Python`, con `Python2`, con `Python3`, con

Python33 e via dicendo.

Un altro carattere speciale è il simbolo `+`. Questo è stretto parente dell'asterisco, poiché identifica una o più occorrenze dell'espressione che lo precede. Un esempio di utilizzo è riportato in [Figura 3.5](#).



Figura 3.5 - Testo che combacia con l'espressione regolare `\d+`, verificato tramite <http://pythex.org>.

Questa è stata una breve introduzione alle espressioni regolari, necessaria, però, per poter comprendere l'esercizio conclusivo. Se intendete approfondire l'argomento potete fare una ricerca su Internet, o, ancora meglio, acquistare dei libri a esse dedicati, come l'ottimo *Espressioni Regolari*, di Marco Beri, scritto, tra l'altro, in lingua italiana.

Il modulo re della libreria standard

In Python il supporto alle espressioni regolari è fornito dal modulo `re` della libreria standard. Ad esempio, la funzione `re.search()` prende come primo argomento un pattern e come secondo argomento la stringa nella quale cercarlo. Se il pattern combacia con del testo all'interno della stringa, allora `re.search()` restituisce un *match object*, ovvero un particolare oggetto che viene valutato `True`:

```
>>> import re
>>> p = re.search('inea', 'primo allineamento')
>>> p
<_sre.SRE_Match object at 0x7f219db9e920>
>>> bool(p)
True
```

Se, invece, il pattern non viene trovato, allora `re.search()` restituisce `None`:

```
>>> p = re.search('inea$', 'primo allineamento')
>>> repr(p)
'None'
```

Vediamo un altro esempio:

```
>>> for line in open('/home/marco/temp/mydir/myfile.txt'):
...     if re.search('inea$', line):
...         print(line)
...
unica linea
```

A questo punto la funzione `file_inspector()` dovrebbe esserci chiara:

```
>>> def file_inspector(file_name: str, pattern: str):
...     for line in open(file_name):
...         if re.search(pattern, line):
...             yield line
...
>>> for line in file_inspector('/home/marco/temp/mydir/myfile.txt', 'inea$'):
...     print(line)
...
unica linea
```

Prende come primo argomento il nome di un file, eventualmente con il percorso completo, come secondo argomento un pattern e restituisce un generatore:

```
>>> g = file_inspector('/etc/group', 'marco$')
>>> next(g)
'adm:x:4:marco\n'
>>> next(g)
'cdrom:x:24:marco\n'
>>> next(g)
'sudo:x:27:marco\n'
```

Per maggiori informazioni sul modulo `re`, possiamo consultare la documentazione online della libreria standard, alla pagina <http://docs.python.org/3/library/re.html>. È disponibile anche un ottimo *how-to* online, alla pagina web <http://docs.python.org/3/howto/regex.html>.

Il parametro `action` di `add_argument()`

A questo punto ci resta da chiarire solamente la seguente linea di codice:

```
parser.add_argument('-r', '--recursive', action='store_true', help='Ricorsiva')
```

Quando da linea di comando viene passato lo switch `-r`, allora ad `args.recursive` viene assegnato `True`, in caso contrario viene assegnato `False`. Questo accade perché abbiamo passato al parametro `action` l'argomento `'store_true'`. Se avessimo passato `action='store_false'` avremmo avuto l'effetto contrario, ovvero l'utilizzo dell'opzione `-r` avrebbe comportato che `args.recursive` sarebbe stato `False`.

Adesso possiamo rileggere lo script *finder.py*. Ci renderemo conto che tutto ciò che inizialmente ci sembrava incomprensibile ora ha un senso.

Moduli, package, ambienti virtuali e applicazioni

Nei capitoli precedenti abbiamo visto in dettaglio il core data type di Python e le funzioni. Concludiamo questa prima parte del libro approfondendo il concetto di **modulo** e introducendo nuovi importanti argomenti. Parleremo di **package** e **scope** nella parte iniziale del capitolo, per poi proseguire affrontando tematiche più pratiche: **ambienti virtuali**, **installazione** e **distribuzione delle applicazioni**, **docstring validation testing**.

Moduli

I *moduli* possono essere utilizzati sia per definire al loro interno attributi accomunati secondo un qualche criterio, consentendo così di strutturare il programma in modo ordinato, o, più semplicemente, per essere eseguiti da linea di comando.

La libreria standard di Python è strutturata in moduli, ciascuno dei quali è sostanzialmente un contenitore di attributi: il modulo `math` contiene classi, funzioni e costanti che forniscono il supporto per la matematica; il modulo `random` fornisce il supporto per la generazione casuale di oggetti, e via dicendo. Come ormai sappiamo, possiamo accedere agli attributi di un modulo, una volta che questo sia stato importato, tramite il delimitatore punto:

```
>>> import math
>>> import random
>>> degrees = random.randrange(360) # Angolo compreso tra 0 e 360
>>> radians = math.radians(degrees) # Convertiamo l'angolo in radianti
>>> math.sin(radians) # Calcoliamo il seno dell'angolo
0.05233595624294381
```

Il nome di un modulo è il nome del file senza il suo suffisso: ad esempio, il file *mymodule.py* è un modulo che si chiama `mymodule`. Un modulo, come ormai sappiamo, viene importato usando l'istruzione `import`. Questa, come prima cosa, cerca il modulo (come descriveremo nella sezione *Percorsi di ricerca dei moduli*); una volta trovato, eventualmente lo compila (come discusso nella sezione *Il flusso di esecuzione dei programmi in Python* del Capitolo 1) e infine lo esegue. Ad esempio, creiamo un modulo `mymodule` contenente una `print()`:

```
$ echo "print('Sto eseguendo una print')"> mymodule.py
```

Come possiamo vedere, l'istruzione `import` causa l'esecuzione del contenuto del modulo:

```
$ python -c "import mymodule"
Sto eseguendo una print
```

Questo è vero solo per il primo `import`, perché per i successivi il modulo non viene nuovamente eseguito:

```
$ python -c "import mymodule; import mymodule; import mymodule"
Sto eseguendo una print
```

Il nome di un modulo ci viene dato dal suo attributo `__name__`:

```
$ python -c "import mymodule; print(mymodule.__name__)"
Sto eseguendo una print
mymodule
```

Tipologie di moduli

I moduli possono appartenere a una delle seguenti categorie: *moduli Python*, *moduli compilati*, *moduli built-in* o estensioni C o C++ compilate e linkate dinamicamente quando importate.

Moduli Python

Abbiamo visto, nella sezione *Introduzione al concetto di modulo* del [Capitolo 1](#), che un modulo Python è semplicemente un file contenente zero (modulo vuoto) o più istruzioni Python. Il seguente modulo `mymodule`, ad esempio, è un modulo vuoto:

```
$ echo "" > mymodule.py
```

Nonostante sia vuoto, se lo importiamo viene compilato e quindi viene prodotto e salvato il suo bytecode:

```
$ python -c "import mymodule"
$ ls
mymodule.py __pycache__
```

Per poter essere importato, un modulo Python deve avere suffisso `.py`:

```
$ echo "print(__name__)" > foomodule
$ python -c "import foomodule"
Traceback (most recent call last):
...
ImportError: No module named 'foomodule'
$ echo "print(__name__)" > foomodule.py
$ python -c "import foomodule"
foomodule
```

Moduli compilati

I moduli compilati, dei quali abbiamo discusso nella sezione *Il flusso di esecuzione dei programmi in Python* del [Capitolo 1](#), sono dei file contenenti

le istruzioni codificate in *bytecode*. Questi moduli vengono salvati su disco quando viene importato il corrispondente modulo Python:

```
$ echo "print(__name__)" > mymodule.py
$ ls
mymodule.py
$ python mymodule.py
__main__
$ ls # La versione compilata del modulo non è stata salvata su disco
mymodule.py
$ python -c "import mymodule" # La versione compilata viene salvata
mymodule
$ ls
mymodule.py __pycache__
```

Per maggiori dettagli, si veda la sezione *La Python Virtual Machine* del Capitolo 1.

Moduli built-in

I moduli built-in sono ottenuti da sorgenti C, compilati e linkati staticamente all'interprete. Il modulo `sys` della libreria standard, ad esempio, è un modulo built-in, per cui, se provassimo a cercare il file `sys.py` nella vostra installazione di Python, non lo troveremmo. Per questo motivo i moduli built-in non hanno l'attributo `__file__`:

```
>>> import os, sys
>>> os.__file__
'/usr/local/lib/python3.4/os.py'
>>> sys.__file__
Traceback (most recent call last):
...
AttributeError: 'module' object has no attribute '__file__'
```

La tupla `sys.builtin_module_names` ha per elementi i nomi dei moduli built-in:

```
>>> import sys
>>> for builtin in sys.builtin_module_names:
...     print(builtin)
...
...
_ast
_codecs
_collections
_functools
_imp
_io
_locale
_sre
```

```
_string
_symtable
_thread
_warnings
_weakref
builtins
errno
faulthandler
gc
itertools
marshal
operator
posix
pwd
signal
sys
xxsubtype
zipimport
```

Le estensioni

Le *estensioni* sono dei moduli ottenuti da sorgenti C o C++ che vengono compilati come librerie dinamiche. Hanno suffisso *.so* su Linux e *.dll* o *.pyd* su Windows. Il modulo `math`, ad esempio, è uno di questi:

```
>>> import math
>>> math.__file__
'/usr/local/lib/python3.4/lib-dynload/math.cpython-34m.so'
```

Dal punto di vista dell'utilizzatore, non vi è alcuna differenza tra queste quattro tipologie di moduli che abbiamo appena visto, poiché tutti i moduli vengono importati e utilizzati allo stesso modo:

```
$ pwd
/usr/local/lib/python3.4
$ find -name "unicodedata*"
./lib-dynload/unicodedata.cpython-34m.so
$ python -c "import unicodedata; print(unicodedata.__name__)"
unicodedata
```

Nel seguito approfondiremo lo studio dei moduli Python.

Tipologie di moduli Python

Come abbiamo già detto, i moduli Python possono avere diverse finalità e modalità di utilizzo: possono essere usati unicamente come contenitori di attributi, oppure solo per essere eseguiti da linea di comando, o anche per entrambi gli scopi assieme. Per poter essere importati, i moduli Python devono avere suffisso *.py*.

Contenitore puro di attributi

Diremo che un modulo Python è un *contenitore puro* quando al suo interno vi sono solo definizioni di attributi (classi, funzioni ecc.). Moduli di questo tipo non vengono eseguiti da linea di comando, ma esclusivamente importati da altri moduli o interattivamente, al fine di utilizzarne gli attributi. Ad esempio, il modulo `numbersgen` è un contenitore puro:

```
$ cat numbersgen.py
"""Definisci alcuni generatori di successioni numeriche."""

def fibonacci(n):
    """Genera numeri di fibonacci."""
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a + b

def wondrous(n):
    """Genera numeri HOTPO (Half Or Triple Plus One)."""
    while n != 1:
        yield n
        n = n // 2 if n%2 == 0 else 3*n + 1
    else:
        yield 1
```

Non ha senso eseguirlo da linea di comando; va solamente importato in modo da utilizzarne gli attributi:

```
>>> import numbersgen
>>> [i for i in numbersgen.fibonacci(14)]
[0, 1, 1, 2, 3, 5, 8, 13]
>>> [i for i in numbersgen.wondrous(14)]
[14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Script

Un modulo Python è detto *script* quando il suo scopo non è quello di essere importato per usare i suoi attributi, ma di essere eseguito al fine di compiere qualche azione. Ad esempio, il file `switch_touchpad.py` è uno script. È facile capire, infatti, che non è stato creato con il fine di utilizzarne gli attributi, ma di essere eseguito da linea di comando, in modo da abilitare o disabilitare il touchpad sulla macchina dell'autore:

```
$ cat switch_touchpad.py
#!/usr/bin/env python
"""Su Ubuntu 11.04: abilita il TouchPad se disabilitato, altrimenti lo disabilita"""
```

```

from os import popen
out = popen('xinput list-props "SynPS/2 Synaptics TouchPad"')
base = 'xinput set-int-prop "SynPS/2 Synaptics TouchPad" "Device Enabled" 8 '
for line in out: # Itera sulle linee dell'output del comando
    line = line.strip()
    if line.startswith("Device Enabled"):
        # Esegui il comando `base + '0'` (disabilita) o `base + '1'` (abilita)
        popen(base + '0') if line.endswith('1') else popen(base + '1')

```

NOTA

La linea di shebang viene utilizzata nei sistemi Unix-like. A partire da Python 3.3 è possibile utilizzare la stessa strategia anche su Windows (PEP-0397). Per maggiori informazioni possiamo consultare la documentazione online: <http://docs.python.org/3/using/windows.html#launcher>.

I moduli ibridi e l'etichetta globale `__name__`

Vi è, infine, una terza categoria di moduli Python che sono allo stesso tempo sia contenitori sia script e che, per tale motivo, chiameremo *moduli ibridi*. Sono dei moduli per i quali tipicamente si vuole che le azioni siano eseguite quando si utilizzano come script, ma non quando li si importa. Si consideri, ad esempio, il modulo *rmfiles.py*:

```

$ cat rmfiles.py
import os

def remove_recursively(from_='.', suffix='.pyc'):
    for root, dirs, files in os.walk(from_):
        found = [file for file in files if file.endswith(suffix)]
        for file in found:
            os.remove(os.path.join(root, file))
        print(os.path.join(root, file), 'file removed')

remove_recursively()

```

Questo definisce una funzione che rimuove in modo ricorsivo, a partire da una certa directory, i file con un dato suffisso. Poiché si vuole che il modulo possa essere direttamente eseguito da linea di comando, in modo che rimuova tutti i file con suffisso *.pyc* a partire dalla directory corrente, dopo la definizione è presente la chiamata `remove_recursively()`. Vediamo un esempio di utilizzo. Creiamo, a tale scopo, cinque file *.pyc* e due file *.py*:

```

$ python -c "import os; [os.popen('touch %d.pyc' %i) for i in range(5)]"
$ python -c "import os; [os.popen('touch %d.py' %i) for i in range(2)]"

```

```
$ ls
0.py 0.pyc 1.py 1.pyc 2.pyc 3.pyc 4.pyc rmfiles_script.py
```

Ecco cosa accade quando usiamo il modulo come script:

```
$ python rmfiles.py # Uso il modulo come script
./2.pyc file removed
./0.pyc file removed
./3.pyc file removed
./1.pyc file removed
./4.pyc file removed
$ ls
0.py 1.py rmfiles.py
```

Supponiamo, ora, di voler creare un nuovo modulo che utilizza la funzione `remove_recursively()` importando `rmfiles`. Questo nuovo modulo è il file *cleaner.py*:

```
$ cat cleaner.py
#!/usr/bin/env python
import argparse
import rmfiles

parser = argparse.ArgumentParser(description='Cancella file in modo ricorsivo')
parser.add_argument('-base_path', type=str, default='.', help='Percorso iniziale')
parser.add_argument('-suffix', type=str, default='.pyc', help='Suffisso dei file')
globals().update(vars(parser.parse_args()))

rmfiles.remove_recursively(base_path, suffix)
```

Eseguiamo lo script *cleaner.py* al fine di cancellare tutti i file *.jpg* dalla directory corrente in giù:

```
$ python cleaner.py -suffix='.jpg'
./generator.pyc file removed
./utils.pyc file removed
./square.pyc file removed
./conf.pyc file removed
./__pycache__/rmfiles.cpython-34.pyc file removed
./img46.jpg file removed
./img45.jpg file removed
./img44.jpg file removed
```

Abbiamo una brutta sorpresa: sono stati cancellati anche i file con suffisso *.pyc*! L'istruzione `import`, infatti, esegue il codice contenuto nel modulo importato, per cui `import rmfiles` ha eseguito il modulo `rmfiles`; questo ha chiamato `remove_recursively()`, la quale ha cancellato tutti i files *.pyc*. Non era affatto ciò che volevamo...

Per risolvere questo problema, si fa ricorso all'utilizzo dell'etichetta globale `__name__`. Quando un modulo *mymodule.py* viene eseguito dalla linea di comando, viene caricato in memoria con il nome `__main__` e non con quello *mymodule*, per cui la sua etichetta `__name__` fa riferimento alla stringa `'__main__'`:

```
$ echo "print(__name__)" > mymodule.py
$ python mymodule.py
__main__
```

Quando, invece, lo stesso modulo viene importato, il suo nome è quello utilizzato per importarlo:

```
$ python -c "import mymodule"
mymodule
```

Poiché, quando eseguiamo *rmfiles.py* da linea di comando, l'etichetta `__name__` fa riferimento a `'__main__'`, mentre quando lo importiamo come modulo fa riferimento a `'rmfiles'`, la soluzione al nostro problema consiste nell'eseguire la chiamata alla funzione `remove_recursively()` solo quando `__name__` è uguale a `'__main__'`. Il file *rmfiles.py* è stato riscritto adottando questa tecnica:

```
$ cat rmfiles.py
#!/usr/bin/env python
"""Rimuovi tutti i file con un dato suffisso.
```

```
Questo modulo definisce la funzione `remove_recursively()`.
Questa prende due argomenti, `from_` e `suffix`, e rimuove a
partire dalla directory `from_`, in modo ricorsivo, tutti i
file che hanno suffisso `suffix`. Per default, rimuove tutti
i file con suffisso `.pyc`, a partire dalla directory corrente.
```

```
Quando il file viene eseguito dalla linea di comando, viene
chiamata la funzione `remove_recursively()` con i valori di
default.
"""
```

```
import os
```

```
def remove_recursively(from_='.', suffix='.pyc'):
    for root, dirs, files in os.walk(from_):
        found = [file for file in files if file.endswith(suffix)]
        for file in found:
            os.remove(os.path.join(root, file))
            print(os.path.join(root, file), 'file removed')
```

```
if __name__ == "__main__":
    remove_recursively()
```

L'istruzione `import rmfiles` non causa più l'esecuzione di `remove_recursively()`:

```
$ ls ~/test
eg.py eg.pyc foo.py foo.pyc
$ python cleaner.py -base_path="$HOME/test/" -suffix=".py"
/home/marco/test/eg.py file removed
/home/marco/test/foo.py file removed
```

Percorsi di ricerca dei moduli

Quando viene eseguita l'istruzione `import mymodule`, l'interprete cerca, prima di tutto, un modulo built-in di nome `mymodule`. Se non lo trova, fa una ricerca all'interno di una serie di percorsi appartenenti alla lista `sys.path`:

```
>>> import sys
>>> type(sys.path) # È una lista
<class 'list'>
>>> for path in sys.path: # I suoi elementi sono stringhe
...     path
...
'/usr/local/lib/python3.4.zip'
'/usr/local/lib/python3.4'
'/usr/local/lib/python3.4/plat-linux'
'/usr/local/lib/python3.4/lib-dynload'
'/usr/local/lib/python3.4/site-packages'
```

La ricerca inizia all'interno del primo percorso di `sys.path` (la stringa vuota corrisponde alla directory corrente), cercando, nell'ordine:

- un *regular package* di nome `mymodule`, ovvero un particolare tipo di modulo che consiste in una directory (in questo caso di nome *mymodule*) contenente un file `__init__.py` (vedremo di che si tratta nella sezione *Regular packages*);
- un modulo Python, ovvero un file sorgente *mymodule.py*;
- un modulo di nome *mymodule.pyc* (vedi PEP-3147);
- un modulo ottenuto da una estensione C o C++ compilata come libreria dinamica, come, ad esempio, i file *mymodule.cpython-34m.so* su Linux, oppure *mymodule.dll* o *mymodule.pyd* su Windows;
- un modulo compilato in bytecode ottimizzato di nome *mymodule.pyo*, ma questo viene cercato solo nel caso in cui l'interprete venga avviato con lo switch `-O`, come mostra il seguente esempio, nel quale il modulo `m` contiene l'istruzione `print(__name__)`:

```
$ ls
m.pyo
$ python -O -c "import m"
m
$ python -c "import m"
Traceback (most recent call last):
...
ImportError: No module named 'm'
```

Appena uno di questi viene trovato, lo si importa e la ricerca termina. Se, invece, non viene importato alcuno dei moduli precedentemente indicati, ma viene trovata una directory di nome *mymodule*, allora questa viene registrata, perché potrebbe diventare un *namespace package* (più avanti in questo capitolo vedremo di cosa si tratta). La ricerca del modulo, a questo punto, continua in modo identico all'interno del percorso successivo di `sys.path`. Se la ricerca termina senza aver importato alcun modulo, si procede nel modo seguente:

- se non è stata registrata alcuna directory di nome *mymodule*, viene lanciata una eccezione `ImportError`;
- se sono state registrate una o più directory di nome *mymodule*, allora Python crea un *namespace package* (un altro tipo particolare di modulo) che comprende tutte queste directory (si veda la sezione *Namespace packages*) e poi lo importa.

Un percorso di ricerca può anche essere un archivio ZIP; in questo caso il modulo all'interno dell'archivio viene automaticamente estratto (vedi PEP-0273 e modulo `zipimport`):

```
$ echo "print(__name__)" > m1.py
$ echo "print(__name__)" > m2.py
$ zip m.zip m*
  adding: m1.py (stored 0%)
  adding: m2.py (stored 0%)
$ ls
m1.py m2.py m.zip
$ rm *.py
$ ls
m.zip
$ python -c "import sys; sys.path.append('$PWD/m.zip'); import m1"
m1
```

Solo i file *.py*, *.pyc* (e *.pyo*, se si utilizza lo switch `-o`) vengono estratti dagli archivi ZIP e importati, ma non i moduli dinamici. Inoltre, Python non modifica l'archivio aggiungendo i corrispondenti *.pyc* (o *.pyo*), per cui, se l'archivio non contiene i file compilati, l'importazione potrebbe essere lenta.

La variabile `sys.path` viene inizializzata nel seguente modo:

- se si sta utilizzando Python in modalità interattiva, viene inserito come primo percorso la directory corrente (stringa vuota), altrimenti la directory contenente lo script:

```
$ pwd
/home/marco/python/repository/python_book
$ echo "import sys; print(sys.path[0])" > ~/test/foo.py
$ python ~/test/foo.py
/home/marco/test
```

- vengono poi aggiunti i percorsi contenuti nella variabile d'ambiente `PYTHONPATH`, definiti utilizzando la medesima sintassi della variabile di shell `PATH`:

```
$ echo $PYTHONPATH
```

```
$ python -c "import sys; {print(path) for path in sys.path[1:]}"
/usr/local/lib/python34.zip
/usr/local/lib/python3.4
/usr/local/lib/python3.4/plat-linux
/usr/local/lib/python3.4/lib-dynload
/usr/local/lib/python3.4/site-packages
$ PYTHONPATH=/base_path1/foo_path1:/base_path2/foo_path2
$ export PYTHONPATH
$ python -c "import sys; {print(path) for path in sys.path[1:]}"
/base_path1/foo_path1
/base_path2/foo_path2
/usr/local/lib/python34.zip
/usr/local/lib/python3.4
/usr/local/lib/python3.4/plat-linux
/usr/local/lib/python3.4/lib-dynload
/usr/local/lib/python3.4/site-packages
```

- vengono aggiunti i percorsi dell'installazione di Python, ricavati utilizzando i percorsi di base `sys.prefix` e `sys.exec_prefix` e completandoli con altri percorsi, come, ad esempio, *Lib/site-packages* su Windows e *lib/pythonX.Y/site-packages* su Unix e Mac:

```
>>> import sys
>>> sys.prefix, sys.exec_prefix
('/usr/local', '/usr/local')
>>> for path in sys.path:
...     if sys.prefix in path:
...         print(path)
...
/usr/local/lib/python34.zip
/usr/local/lib/python3.4
```

```
/usr/local/lib/python3.4/plat-linux
/usr/local/lib/python3.4/lib-dynload
/usr/local/lib/python3.4/site-packages
```

- se, all'interno dei percorsi di installazione di Python discussi al punto precedente, ci sono dei file con suffisso *.pth* aventi come linee dei percorsi, allora anche tali percorsi, dopo che la loro esistenza sia stata verificata, vengono aggiunti a `sys.path`. Solitamente i files *.pth* vengono posti nella directory *.../site-packages/*:

```
$ PYSITE=/usr/local/lib/python3.4/site-packages
$ # Su $PYSITE attualmente non esistono file con suffisso .pth
$ ls $PYSITE/*.pth 2> /dev/null
$ python -c "import sys; {print(path) for path in sys.path[1:]}"
/usr/local/lib/python34.zip
/usr/local/lib/python3.4
/usr/local/lib/python3.4/plat-linux
/usr/local/lib/python3.4/lib-dynload
/usr/local/lib/python3.4/site-packages
$ PYPATH="$HOME/test\n$HOME/.vim"
$ # Creo un file foo.pth in $PYSITE, contenente due percorsi
$ echo -e $PYPATH | sudo tee $PYSITE/foo.pth
/home/marco/test
/home/marco/.vim
$ # Verifico che il file foo.pth sia stato creato
$ ls $PYSITE/*.pth 2> /dev/null
/usr/local/lib/python3.4/site-packages/foo.pth
# I percorsi indicati in foo.pth sono stati inseriti in sys.path
$ python -c "import sys; {print(path) for path in sys.path[1:]}"
/usr/local/lib/python34.zip
/usr/local/lib/python3.4
/usr/local/lib/python3.4/plat-linux
/usr/local/lib/python3.4/lib-dynload
/usr/local/lib/python3.4/site-packages
/home/marco/test
/home/marco/.vim
```

La variabile `sys.path` può essere modificata dopo la sua inizializzazione, per estendere i percorsi di ricerca in modo dinamico:

```
>>> from os.path import join
>>> root = '/home/marco/temp'
>>> name = 'mymodule.py'
>>> open(join(root, name), 'w').write('print("I am", __name__)')
23
>>> import sys
>>> root in sys.path
False
>>> import mymodule # Il modulo appena creato non viene trovato
Traceback (most recent call last):
...
ImportError: No module named 'mymodule'
```



```
>>> sys.path.append(root) # Aggiungiamo il percorso di `mymodule.py`
>>> root in sys.path
True
>>> import mymodule # Adesso Python riesce a trovare `mymodule`
I am mymodule
```

Varianti dell'istruzione import

È possibile importare un modulo utilizzando un nome alternativo, estendendo l'istruzione `import` con la parola chiave `as`:

```
>>> import encodings as enc
```

Adesso è l'etichetta `enc` a fare riferimento al modulo `encodings`, e l'etichetta `encodings` non è definita:

```
>>> enc.__name__
'encodings'
>>> encodings.__name__ # Infatti...
Traceback (most recent call last):
...
NameError: name 'encodings' is not defined
```

Sostanzialmente, l'istruzione `import encodings as enc` è equivalente al seguente codice:

```
>>> import encodings
>>> enc = encodings
>>> del encodings
```

L'istruzione `import` può essere utilizzata anche in combinazione con la parola chiave `from`, in modo da accedere direttamente agli attributi di un modulo, senza la necessità di utilizzare il nome del modulo e il delimitatore punto:

```
>>> from sys import platform
>>> platform # Evito così di utilizzare la notazione `sys.platform`
'linux'
```

Si osservi che, anche questa volta, è stata copiata solo l'etichetta `platform`, per cui `sys` non è definita:

```
>>> sys
Traceback (most recent call last):
...
NameError: name 'sys' is not defined
```

Anche questa forma può essere estesa con la parola chiave `as`:

```
>>> from sys import getrecursionlimit as rlimit
>>> rlimit() # Anzichè getrecursionlimit()
1000
```

L'ultima variante è `from mymodule import *`, la quale importa tutte le etichette definite nel modulo `mymodule`:

```
>>> from os import *
>>> getlogin() # `getlogin` è un attributo del modulo `os`
'marco'
```

A differenza delle altre varianti di `import`, che possono essere annidate all'interno di qualsiasi blocco, `from mymodule import *` può essere usata solo a livello di modulo:

```
>>> def foo():
...     from os import *
...
File "<stdin>", line 1
SyntaxError: import * only allowed at module level
```

Se vogliamo che l'istruzione `from mymodule import *` non importi alcune etichette, basta far iniziare queste ultime con un underscore. Consideriamo, ad esempio, il seguente modulo:

```
$ cat mymodule.py
_a = 33
aa = 44
```

Ecco cosa accade quando importiamo tutte le etichette:

```
>>> from mymodule import * # Non importa le etichette che iniziano con `_`
>>> aa
44
>>> _a # `_a` inizia per underscore, quindi non è stata importata
Traceback (most recent call last):
...
NameError: name '_a' is not defined
>>> from mymodule import _a
>>> _a
33
```

Un altro modo per stabilire quali etichette importare con `from mymodule import *` è quello di definire in `mymodule` una lista (o anche una tupla) assegnata all'etichetta `__all__`, avente per elementi i nomi delle etichette da importare. Ad esempio, se consideriamo il seguente modulo:

```
$ cat mymodule.py
__all__ = ['a', 'c']
a = 1
b = 2
c = (1, 2, 3)
```

possiamo importare con `from mymodule import *` solo `a` e `c`, ma non `b`:

```
>>> from mymodule import *
>>> a
1
>>> c
(1, 2, 3)
>>> b
Traceback (most recent call last):
...
NameError: name 'b' is not defined
```

Talvolta può essere utile importare un modulo dato il suo nome sotto forma di stringa. Per fare ciò si utilizzano le funzioni `importlib.__import__()` o `importlib.import_module()`:

```
>>> import importlib
>>> s = importlib.__import__('sys')
>>> import sys
>>> s is sys
True
>>> s.platform
'linux'
>>> enc = importlib.import_module('encodings')
>>> enc.__name__
'encodings'
```

NOTA

Per importare un modulo, dato il suo nome sotto forma di stringa, si può utilizzare anche la funzione built-in `__import__()`.

Questo ci consente di importare un modulo il cui nome ci sarà noto solo quando il programma è in esecuzione:

```
>>> while True:
...     module_name = input("Inserisci il nome del modulo: ")
...     mymodule = importlib.__import__(module_name)
...     print('Hai importato il modulo', mymodule.__name__)
...
```

```
Inserisci il nome del modulo: os
Hai importato il modulo os
Inserisci il nome del modulo: encodings
Hai importato il modulo encodings
Inserisci il nome del modulo: non_saprei
Traceback (most recent call last):
...
ImportError: No module named 'non_saprei'
```

La differenza principale tra `importlib.__import__()` e `importlib.import_module()` è che quest'ultima restituisce il package più annidato, mentre la prima restituisce il package di livello più alto:

```
>>> import importlib
>>> a = importlib.import_module('root.nested')
>>> a.__name__
'root.nested'
>>> a = importlib.__import__('root.nested')
>>> a.__name__
'root'
```

NOTA

Per maggiori informazioni sull'importazione dei moduli possiamo consultare la documentazione online del modulo `importlib`.

Concludiamo questa parte con un'osservazione. Se importiamo un modulo con `import mymodule` e successivamente importiamo direttamente una etichetta dal modulo con `from mymodule import etichetta`, un assegnamento a una delle due non avrà effetto sull'altra:

```
>>> import mymodule
>>> mymodule.n, mymodule.mylist
(44, [1, 2, 3])
>>> mymodule.mylist = [1, 2, 3, 4]
>>> mymodule.n = 33
>>> from mymodule import n, mylist
>>> n, mylist
(33, [1, 2, 3, 4])
>>> mymodule.mylist = [1, 2, 3, 4, 5]
>>> mylist # L'etichetta 'mylist' fa ancora riferimento al vecchio oggetto
[1, 2, 3, 4]
```

Infatti `mymodule.etichetta` ed `etichetta` sono due etichette diverse che fanno riferimento al medesimo oggetto, esattamente come accade nel seguente esempio per `a` e `b`:

```
>>> a = b = [1, 2, 3]
>>> a.append(4) # Modifico l'oggetto
>>> a
[1, 2, 3, 4]
>>> b
[1, 2, 3, 4]
>>> a = (1, 2, 3) # Assegno all'etichetta `a` un nuovo oggetto
>>> a
(1, 2, 3)
>>> b # L'etichetta `b` continua a fare riferimento al vecchio oggetto
[1, 2, 3, 4]
```

Importazione di moduli già importati e funzione `imp.reload()`

Consideriamo il seguente modulo:

```
$ cat mymodule.py
print(__name__)
mylist = [1, 2, 3]
```

L'istruzione `import`, in tutte le sue forme, esegue il modulo solo la prima volta che questo viene importato:

```
>>> from mymodule import mylist
mymodule
>>> import mymodule
>>> import mymodule as mym
```

L'importazione di un modulo, infatti, è un'operazione dispendiosa, così Python per default la compie solo la prima volta, mentre gli `import` successivi semplicemente recuperano il modulo già caricato in memoria.

I moduli già caricati sono memorizzati su un dizionario, `sys.modules`, il quale ha come chiavi i nomi dei moduli e come valori i riferimenti ai moduli stessi:

```
>>> import sys
>>> 'mymodule' in sys.modules
False
>>> import mymodule
mymodule
>>> 'mymodule' in sys.modules
True
```

Se un valore di `sys.modules` viene impostato a `None`, allora l'importazione del modulo causerà il lancio di una eccezione `ImportError`:

```
>>> sys.modules['mymodule'] = None
>>> import mymodule
Traceback (most recent call last):
```

```
...
ImportError: import of 'mymodule' halted; None in sys.modules
```

Se il nome di un modulo non è presente in `sys.modules`, allora Python è costretto a eseguire l'operazione di `import`, effettuando quindi la ricerca del modulo e la sua esecuzione:

```
>>> import sys
>>> import mymodule
mymodule
>>> del sys.modules['mymodule']
>>> import mymodule # Viene eseguito nuovamente
mymodule
```

Una conseguenza del fatto che i moduli vengono importati solo una volta è che, se si importa un modulo e poi si modifica un suo attributo, quando il modulo viene nuovamente importato, il valore dell'attributo non viene inizializzato. Ecco ciò che accade quando si modifica l'oggetto riferito da `mymodule.mylist` e poi si importa nuovamente `mymodule`:

```
>>> import mymodule
mymodule
>>> mymodule.mylist
[1, 2, 3]
>>> mymodule.mylist.append(4)
>>> mymodule.mylist
[1, 2, 3, 4]
>>> import mymodule # Il modulo non viene eseguito
>>> mymodule.mylist # Infatti `mylist` non viene inizializzata a [1, 2, 3]
[1, 2, 3, 4]
```

Si può inizializzare il modulo ricaricandolo mediante la funzione `imp.reload()`:

```
>>> from imp import reload
>>> reload(mymodule) # Il modulo viene eseguito
mymodule
<module 'mymodule' from './mymodule.py'>
>>> mymodule.mylist # Infatti `mylist` è stata inizializzata
[1, 2, 3]
```

Import circolari

Quando due moduli si importano a vicenda, si dice che si ha un *import circolare* (*circular import*). Gli import circolari sono causa, talvolta, di problemi la cui risoluzione non è immediata. Consideriamo, ad esempio, i seguenti due moduli:

```
$ ls
main.py mymodule.py
```

Il modulo `main` stampa il suo nome, poi importa `mymodule` e infine assegna la stringa `'main.py'` all'etichetta `a`:

```
$ cat main.py
print(__name__)
import mymodule
a = 'main.py'
```

Il modulo `mymodule`, invece, stampa il suo nome, assegna la stringa `'mymodule.py'` all'etichetta `a`, importa il modulo principale del programma e infine stampa l'etichetta `a` di quest'ultimo:

```
$ cat mymodule.py
print(__name__)
a = 'mymodule.py'
import __main__
print(__main__.a)
```

Ecco cosa accade se eseguiamo *mymodule.py* da linea di comando:

```
$ python mymodule.py
__main__
mymodule.py
```

Viene stampato il nome del modulo, che è `'__main__'`, visto che *mymodule.py* è stato eseguito da linea di comando. Poi viene eseguita l'istruzione `import __main__`, la quale, però, non importa `__main__` poiché questo è già in esecuzione e quindi presente su `sys.modules`; l'unico effetto di `import __main__` è quello di rendere disponibile l'etichetta `__main__`. Viene infine stampata `__main__.a`, ovvero la stringa `'mymodule.py'` assegnata all'etichetta `a` due istruzioni sopra la `print()`.

Ecco, invece, cosa accade se eseguiamo *main.py* da linea di comando:

```
$ python main.py
__main__
mymodule
Traceback (most recent call last):
...
AttributeError: 'module' object has no attribute 'a'
```

Viene stampato il nome del modulo, che anche questa volta è `'__main__'`, visto che *main.py* è stato eseguito da linea di comando. Fatto ciò, però, prima che venga assegnata l'etichetta `a` viene importato `mymodule`, e questa istruzione

comporta l'esecuzione del codice contenuto in *mymodule.py*, ovvero:

1. `print(__name__)`: stampa il nome del modulo, ovvero `mymodule`;
2. `a = 'mymodule.py'`: assegna la stringa `'mymodule.py'` all'etichetta `a` nel namespace di `mymodule`;
3. `import __main__`: il modulo `__main__` (associato al file eseguito da linea di comando, *main.py*) è già presente in `sys.modules`, quindi non viene importato, e questa istruzione ha il solo effetto di rendere disponibile l'etichetta `__main__` nel modulo `mymodule`;
4. `print(__main__.a)`: questa istruzione dovrebbe stampare `__main__.a`. Quando, però, Python prova a risolvere l'etichetta `a` nel namespace del modulo principale, non la trova. Infatti questa etichetta non è stata ancora definita, poiché il flusso di esecuzione in *main.py* è ancora fermo alla penultima istruzione, `import mymodule`.

Per evitare questo genere di problemi, possiamo adottare le pratiche elencate alla pagina *Programming FAQ* del sito ufficiale: <http://docs.python.org/3/faq/programming.html>, alla sezione intitolata *What are the “best practices” for using import in a module?*.

Concludiamo questa sezione con una osservazione in merito al modulo `__main__`. Quando un file *mymodule.py* viene eseguito da linea di comando, il suo contenuto viene caricato in memoria e associato a un modulo chiamato `__main__`. Quindi è quest'ultimo a essere presente in `sys.modules` e non `mymodule`. Se, pertanto, in *mymodule.py* avessimo importato `main` piuttosto che `__main__`, il problema non si sarebbe presentato:

```
$ cat mymodule.py
print(__name__)
a = 'mymodule.py'
import main
print(main.a)
$ python main.py
__main__
mymodule
main
main.py
```

Questo perché, quando il modulo `main.py` viene eseguito da linea di comando, il suo codice viene caricato in memoria e assegnato al modulo `__main__`, quindi è questo a essere presente in `sys.modules` e non `main`:

```
$ cat mymodule.py
print(__name__)
```



```

a = 'mymodule.py'
import sys
print("__main__" in sys.modules?", '__main__' in sys.modules)
print("main" in sys.modules?", 'main' in sys.modules)
import main
print("Dopo l'import: 'main' in sys.modules?", 'main' in sys.modules)
print(main.a)
$ python main.py
__main__
mymodule
'__main__' in sys.modules? True
'main' in sys.modules? False
main
Dopo l'import: 'main' in sys.modules? True
main.py

```

In questo caso, quindi, l'istruzione `import main`, a differenza di prima (quando c'era `import __main__`), importa realmente il modulo `main`. Le sue istruzioni vengono eseguite in *main.py*, sino ad arrivare a `import mymodule`, la quale, però, non importa `mymodule`, visto che questo è già presente in `sys.modules`. Infine, viene assegnata l'etichetta `a`, questa volta in *mymodule.py*.

Package

I *package* sono oggetti di tipo `module` aventi l'attributo `__path__`:

```

>>> import os
>>> os.mkdir('pk') # Creo la directory `pk`
>>> import pk # Posso importare `pk`
>>> type(pk), type(os)
(<class 'module'>, <class 'module'>)
>>> hasattr(pk, '__path__') # `pk` è un package
True
>>> pk.__path__
_NamespacePath(['./pk'])
>>> hasattr(os, '__path__') # `os` non è un package
False

```

Quindi tutti i *package* sono moduli, ma non tutti i moduli sono *package*. Possiamo pensare al *package* come a un contenitore di moduli; l'analogia è azzeccata poiché, tipicamente, un *package* viene costruito a partire da una o più *directory* che si trovano nei percorsi di ricerca di Python. I *package* ci consentono di distinguere moduli diversi aventi lo stesso nome. Infatti, se non utilizziamo i *package*, quando due moduli con lo stesso nome si trovano entrambi nei percorsi di ricerca di Python, il fatto che venga importato uno piuttosto che l'altro dipende solo dall'ordine dei percorsi di ricerca in `sys.path`. Può quindi accadere che venga importato il modulo sbagliato, poiché questo

viene trovato da Python prima di quello che si vuole realmente importare:

```
$ mkdir foo moo # Creiamo due directory `foo` e `moo`
$ # Aggiungiamo alle directory due moduli con lo stesso nome
$ echo "print(__file__)" > foo/mymodule.py # Questo modulo stampa il percorso del file
$ echo "print(__file__)" > moo/mymodule.py # Questo modulo stampa il percorso del file
$ PYTHONPATH=$PWD/foo:$PWD/moo # Inseriamo le directory nei percorsi di ricerca
$ export PYTHONPATH
$ # Vorremmo importare `./moo/mymodule.py`, ma verrà importato `./foo/mymodule.py`
$ python -c "import mymodule"
/home/marco/test/foo/mymodule.py
$ # Infatti la directory `foo` viene ispezionata da Python prima di `moo`
$ python -c "import sys; {print(p) for p in sys.path[:3]}"
```

```
/home/marco/test/foo
/home/marco/test/moo
```

Utilizzando i package risolviamo il problema, poiché possiamo accedere al modulo corretto qualificandolo mediante il nome del package:

```
$ python -c "import moo.mymodule" # Accediamo al modulo tramite `moo`
./moo/mymodule.py
$ python -c "import foo.mymodule" # Accediamo al modulo tramite `foo`
./foo/mymodule.py
```

Utilizzo dei package

Come abbiamo appena visto, è possibile navigare tra i package con il delimitatore punto:

```
$ mkdir root_dir # Creiamo la directory `root_dir`
$ mkdir root_dir/nested_dir # Creiamo `nested_dir` interna a `root_dir`
$ # Creiamo un modulo `m` all'interno del package `nested_dir`
$ echo "print(__name__)" > root_dir/nested_dir/m.py
$ # Possiamo importare `m` navigando tra i packages
$ python -c "import root_dir.nested_dir.m"
root_dir.nested_dir.m
```

Poiché un package è un oggetto di tipo `module`, l'istruzione `import` estesa con la parola chiave `from` consente di importare sia i moduli semplici attraverso i package, sia i package stessi:

```
>>> from root_dir.nested_dir import m
root_dir.nested_dir.m
>>> from root_dir import nested_dir
>>> nested_dir.__path__
_NamespacePath(['./root_dir/nested_dir'])
```

Tutti i package intermedi attraversati tramite il delimitatore punto vengono anch'essi importati ed è possibile, quindi, accedere ai loro attributi:

```
>>> import root_dir.nested_dir.m
root_dir.nested_dir.m
>>> root_dir.nested_dir.__package__
'root_dir.nested_dir'
```

L'attributo `__package__` di un modulo corrisponde al nome del package tramite il quale è stato importato:

```
>>> root_dir.nested_dir.m.__package__
'root_dir.nested_dir'
>>> import os
>>> os.__package__
''
```

Un package, per poter essere importato, deve trovarsi all'interno dei percorsi di ricerca di Python. Se un package viene importato mediante altri package, è sufficiente che il package più esterno si trovi nei percorsi di ricerca. Nel nostro esempio, infatti, `root_dir` si trova all'interno della directory dalla quale abbiamo avviato l'interprete, che rappresenta il primo percorso di ricerca:

```
>>> import os
>>> import sys
>>> os.path.abspath(sys.path[0]) # Percorso assoluto di `sys.path[0]`
'/home/marco/test'
>>> os.path.abspath(root_dir.__path__[0])
'/home/marco/test/root_dir'
```

I package possono essere di due tipi: *regular package* e *namespace package*.

Regular package

Un regular package è tipicamente una directory contenente un file `__init__.py`, il quale viene importato quando si importa il package. Se più regular package vengono attraversati in una operazione di `import`, tutti i loro moduli `__init__` vengono importati in cascata:

```
>>> # Creiamo nelle directory `root_dir` e `nested_dir` i files `__init__.py`
... f1 = open('root_dir/__init__.py', 'w')
>>> f2 = open('root_dir/nested_dir/__init__.py', 'w')
>>> # Adesso le due directory sono dei regular package
... f1.writelines(['print(__name__)\n', 'num = 10\n'])
>>> f2.writelines(['print(__name__)\n', 'num = 100\n'])
>>> f1.close()
```

```
>>> f2.close()
>>> import root_dir.nested_dir # Esegue tutti gli `__init__.py`
root_dir
root_dir.nested_dir
```

Importare un regular package significa quindi, sostanzialmente, importare il suo modulo `__init__`:

```
>>> root_dir.nested_dir.__file__
'./root_dir/nested_dir/__init__.py'
>>> root_dir.__cached__
'./root_dir/__pycache__/__init__.cpython-34.pyc'
```

Gli attributi del regular package sono quelli definiti nel modulo `__init__`:

```
>>> 'num' in root_dir.__dict__ # Namespace implementato con dizionario
True
>>> root_dir.num
10
>>> root_dir.nested_dir.num
100
```

I moduli contenuti all'interno del regular package, invece, non fanno parte dei suoi attributi:

```
>>> hasattr(root_dir.nested_dir, 'm')
False
>>> [n for n in dir(root_dir.nested_dir) if not n.startswith('__')]
['num']
```

Diventano, però, attributi nel momento in cui vengono importati tramite il package:

```
>>> import root_dir.nested_dir.m
root_dir.nested_dir.m
>>> hasattr(root_dir.nested_dir, 'm')
True
>>> [n for n in dir(root_dir.nested_dir) if not n.startswith('__')]
['m', 'num']
```

Il fatto che un modulo diventi attributo di un package solo se esplicitamente importato consente di evitare che, importando un package, tutti i moduli in esso contenuti vengano automaticamente importati. Questo causerebbe, infatti, un grosso dispendio di risorse. Per gli stessi motivi, anche l'istruzione `from package import *` importa solamente gli attributi definiti nel modulo `__init__` e il package stesso, ma non i moduli contenuti nel package:

```
>>> from root_dir.nested_dir import *
root_dir
root_dir.nested_dir
>>> num
100
>>> m
Traceback (most recent call last):
...
NameError: name 'm' is not defined
```

Se si vuole che alcuni moduli contenuti nel package vengano importati con l'istruzione `from package import *`, è possibile elencarli in una lista o tupla assegnata all'etichetta `__all__`, definita dal modulo `__init__` del package stesso:

```
$ # Creiamo 3 moduli nel package `root_dir`
$ echo "print(__name__)" > root_dir/a.py
$ echo "print(__name__)" > root_dir/b.py
$ echo "print(__name__)" > root_dir/c.py
$ # Non vengono importati con `from package import *`
$ python -c "from root_dir import *"
$ # Creiamo l'attributo `__all__` su `root_dir/__init__.py`
$ echo "__all__ = ['a', 'c']" >> root_dir/__init__.py
$ # Adesso i moduli `a` e `c` vengono importati, ma `b` no
$ python -c "from root_dir import *"
root_dir.a
root_dir.c
```

Namespace package

I *namespace package* sono stati introdotti a partire da Python 3.3 (vedi PEP-0420). Come abbiamo visto nella sezione *Percorsi di ricerca dei moduli*, quando viene eseguita l'istruzione `import foo`, finché non viene trovato un modulo (compresi i regular package) con quel nome, tutte le directory `foo` che si trovano nei percorsi di ricerca vengono registrate e, se la ricerca termina senza aver importato alcun modulo, allora con le eventuali directory registrate viene creato implicitamente un package, detto *implicit namespace package*, o, più semplicemente, *namespace package*. Ad esempio, creiamo la seguente gerarchia di directory:

```
$ mkdir foo aoo boo
$ mkdir aoo/foo boo/coo boo/coo/foo
$ echo "print(__file__)" > foo/m.py
$ echo "print(__file__)" > aoo/foo/a.py
$ echo "print(__file__)" > boo/coo/foo/b.py
$ ls -R
.:
aoo boo foo
```

```
./aoo:
```

foo

./aoo/foo:
a.py

./boo:
coo

./boo/coo:
foo

./boo/coo/foo:
b.py

./foo:
m.py

Aggiungiamo al `PYTHONPATH` le directory che contengono *foo*:

```
$ PYTHONPATH=$PWD/aoo:$PWD/boo/coo  
$ export PYTHONPATH
```

Se ora importiamo `foo`, poiché Python non trova alcun modulo con questo nome, crea un modulo di tipo namespace package con tutte le directory *foo* che ha trovato nei percorsi di ricerca:

```
>>> import foo  
>>> type(foo)  
<class 'module'>  
>>> foo.__path__  
_NamespacePath(['./foo', '/home/marco/temp/aoo/foo', '/home/marco/temp/boo/coo/foo'])  
>>> from foo import m  
./foo/m.py  
>>> from foo import a  
/home/marco/temp/aoo/foo/a.py  
>>> from foo import b  
/home/marco/temp/boo/coo/foo/b.py
```

A differenza dei regular package e di tutti gli altri moduli, i namespace package non hanno l'attributo `__file__` e, di conseguenza, neppure l'attributo `__cached__`:

```
>>> hasattr(foo, '__file__'), hasattr(foo, '__cached__')  
(False, False)
```

I namespace package si comportano allo stesso modo dei regular package; anche in questo caso, i moduli che li compongono, chiamati *sezioni*, non

fanno parte dei loro attributi, a meno che non vengano esplicitamente importati:

```
>>> import foo
>>> hasattr(foo, 'm')
False
>>> from foo import *
>>> m
Traceback (most recent call last):
...
NameError: name 'm' is not defined
>>> from foo import m
./foo/m.py
>>> hasattr(foo, 'm')
True
```

Percorsi relativi all'interno dei package

Un modulo importato tramite un package può utilizzare dei percorsi relativi per importare dei moduli interni al package:

```
$ mkdir top # Creiamo una directory `top`
$ echo "print(__file__)" > top/mymodule.py # Creo `mymodule.py` in `top`
$ # Creiamo un modulo `mymain` che importa con un percorso relativo `mymodule`
$ echo "from . import mymodule" > top/mymain.py
$ python -B -c "import top.mymain" # Ecco che `mymain` importa `mymodule`
./top/mymodule.py
```

I moduli importati tramite i package possono utilizzare i percorsi relativi anche per importare dei sotto-package interni al package di livello più alto:

```
$ # Creiamo due package interni al package `top`
$ mkdir top/nested1 top/nested2
$ touch top/nested1/__init__.py top/nested2/__init__.py
$ # Aggiungiamo una `print` al file `__init__.py` del package `nested2`
$ echo "print(__file__)" >> top/nested2/__init__.py
$ # Aggiungiamo a `__init__.py` di `nested1` un import relativo a `nested2`
$ echo "from .. import nested2" >> top/nested1/__init__.py
$ ls -R # Ecco come si presenta la gerarchia dei packages
.:
top
```

```
./top:
mymain.py mymodule.py nested1 nested2
```

```
./top/nested1:
__init__.py
```

```
./top/nested2:
__init__.py
```

```
$ python -B -c "import top.nested1" # Viene importato `nested2`  
./top/nested2/__init__.py
```

Si possono importare anche i moduli di un altro sotto-package:

```
$ # Creiamo il modulo `mymodule` nel package `nested2`  
$ echo "print(__file__)" > top/nested2/mymodule.py  
$ # `__init__.py` di `nested1` importa `mymodule` di `nested2`  
$ echo "from ..nested2 import mymodule" >> top/nested1/__init__.py  
$ ls -R  
.:  
top
```

```
./top:  
mymain.py mymodule.py nested1 nested2
```

```
./top/nested1:  
__init__.py
```

```
./top/nested2:  
__init__.py mymodule.py  
$ python -B -c "import top.nested1"  
./top/nested2/__init__.py  
./top/nested2/mymodule.py
```

Tutti i percorsi devono, però, essere interni al package di livello più alto:

```
$ mkdir top_level # Creiamo un package allo stesso livello di `top`  
$ ls  
top top_level  
$ # Creiamo il modulo `mymodule` nel package `top_level`  
$ echo "print(__file__)" > top_level/mymodule.py  
$ # In `mymain` di `top` scriviamo un import relativo esterno a `top`  
$ echo "from ..top_level import mymodule" >> top/mymain.py  
$ python -B -c "import top.mymain" # L'import esterno non viene eseguito  
./top/mymodule.py  
Traceback (most recent call last):  
...  
ValueError: attempted relative import beyond top-level package
```


Namespace, scope e risoluzione dei nomi

Diciamo che un'etichetta è *visibile* in un certo punto del programma se in quel punto è possibile inserirla all'interno di un'espressione. Utilizziamo, invece, il termine *namespace* per fare riferimento a un contenitore astratto di etichette. Le etichette appartenenti a differenti namespace non hanno alcuna relazione tra di loro, nel senso che due namespace distinti possono avere etichette uguali che fanno riferimento a oggetti differenti.

In Python un'etichetta può appartenere a uno dei seguenti namespace: *built-in*, *globale* o *locale*. Questi tre namespace vengono creati in momenti diversi durante l'esecuzione del programma e hanno cicli di vita differenti.

Il namespace built-in

Il *namespace built-in* viene creato nel momento in cui l'interprete viene avviato e non viene più cancellato per tutta la vita del programma. Ciò significa che le etichette appartenenti a questo namespace sono visibili ovunque e in qualunque momento. Le funzioni e le classi built-in, delle quali abbiamo discusso nei capitoli precedenti, appartengono a questo namespace. Le etichette appartenenti al namespace built-in sono dette *etichette built-in* e sono dei riferimenti agli attributi del modulo `builtins`:

```
>>> import builtins
>>> for name in dir(builtins):
...     print(name)
...
ArithmeticError
AssertionError
AttributeError
...
IndentationError
IndexError
...
TypeError
UnboundLocalError
...
abs
all
any
...
tuple
type
vars
```

zip

Il namespace built-in è implementato tramite un oggetto di tipo dizionario, `builtins.__dict__`:

```
>>> type(builtins.__dict__)
<class 'dict'>
>>> 'sum' in builtins.__dict__
True
>>> builtins.__dict__['sum']
<built-in function sum>
>>> sum is builtins.__dict__['sum']
True
>>> builtins.__dict__['sum']([1, 2, 3, 4])
10
```

Questo oggetto mappa le etichette (chiavi) nei corrispondenti riferimenti agli oggetti (valori).

Namespace globale

Si dice che le parti di codice di un modulo che si trovano all'esterno di ogni definizione di classe (istruzione `class`) o di funzione (istruzione `def`) appartengono al livello alto del modulo (*top-level*).

Il *namespace globale* di un modulo è l'insieme delle etichette create al livello alto del modulo:

```
>>> a = 33 # L'etichetta `a` appartiene al namespace globale
>>> def foo(): # L'etichetta `foo` appartiene al namespace globale
... pass
...
>>> class Foo: # L'etichetta `Foo` appartiene al namespace globale
... pass
...
>>> for i in range(3):
...     print(i)
...
0
1
2
>>> i # L'etichetta `i` è globale
2
>>> if True:
...     b = 33
...
>>> b # Anche `b` è globale
33
```

Una etichetta appartenente al namespace globale viene detta *globale*. Quando gli assegnamenti e le definizioni vengono fatti in modo interattivo, come nel precedente esempio, vengono create delle etichette globali a un modulo chiamato `__main__`:

```
>>> foo.__module__, Foo.__module__, __name__  
('__main__', '__main__', '__main__')
```

La stessa cosa si verifica nel caso in cui il modulo venga eseguito come script:

```
$ echo -e "def foo(): pass\nprint(foo.__module__)" > mymodule.py  
$ python mymodule.py  
__main__  
$ python -m mymodule  
__main__
```

Se, invece, il modulo viene importato, allora tutti i suoi attributi avranno come attributo `__module__` il nome del modulo importato:

```
$ echo "def foo(): pass" > mymodule1.py  
$ echo "from mymodule1 import foo; print(foo.__module__)" > mymodule2.py  
$ python mymodule2.py  
mymodule1
```

Il namespace globale di un modulo è implementato mediante l'attributo `__dict__` del modulo, un dizionario avente come chiavi delle stringhe che rappresentano i nomi delle etichette e come valori i riferimenti agli oggetti:

```
>>> # Creiamo un modulo di nome `mymodule`  
... open('mymodule.py', 'w').write('print(__name__)')  
15  
>>> import mymodule  
mymodule  
>>> 'a' in mymodule.__dict__  
False  
>>> mymodule.a = 44  
>>> 'a' in mymodule.__dict__  
True  
>>> mymodule.__dict__['a']  
44  
>>> mymodule.__dict__['a'] = 33  
>>> mymodule.a  
33  
>>> mymodule.__dict__['b'] = [1, 2, 3]  
>>> 'b' in mymodule.__dict__  
True  
>>> mymodule.b  
[1, 2, 3]
```

Quindi possiamo fare riferimento a una etichetta definita in un modulo `mymodule` sia con la notazione `mymodule.etichetta` sia tramite il dizionario `mymodule.__dict__`, utilizzando in questo caso come chiave una stringa che rappresenta il nome dell'etichetta: `mymodule.__dict__['etichetta']`. Approfondiremo il discorso nel [Capitolo 6](#), quando parleremo dei descriptor.

La funzione built-in `globals()` restituisce il namespace globale del modulo:

```
>>> globals
<built-in function globals>
>>> globals()
{'__name__': '__main__', '__doc__': None, '__builtins__': <module 'builtins'>, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>}
>>> globals().keys()
dict_keys(['__name__', '__doc__', '__builtins__', '__package__', '__loader__'])
>>> import importlib
>>> m = importlib.import_module(__name__)
>>> m.__name__ # Il modulo importato è __main__
'__main__'
>>> m.__dict__ is globals() # Restituisce il namespace, non una copia
True
```

L'etichetta globale `__builtins__` fa riferimento al modulo `builtins`:

```
>>> __builtins__
<module 'builtins'>
>>> import builtins
>>> builtins is __builtins__
True
```

Dopo che un modulo è stato importato, il suo namespace continua a esistere per tutta la vita del programma, anche se il modulo non ha più etichette che fanno riferimento a esso:

```
>>> import sys
>>> 'math' in sys.modules # `sys.modules` è il dizionario dei moduli importati
False
>>> import math
>>> 'math' in sys.modules
True
>>> math.a = 44
>>> sys.modules['math'].a
44
>>> del math
>>> 'math' in sys.modules # Il modulo `math` è ancora in memoria
True
>>> 'a' in sys.modules['math'].__dict__ # Quindi anche il suo namespace globale
True
>>> sys.modules['math'].a
44
```

Namespace locale

Il *namespace locale* è l'insieme delle etichette assegnate implicitamente o esplicitamente in una funzione, in un metodo, in una classe, oppure in una comprehension. Viene creato nel momento in cui il codice viene eseguito e, nel caso delle funzioni, dei metodi e delle comprehension, e viene cancellato al termine dell'esecuzione. Una etichetta appartenente a un namespace locale è detta *locale*.

La funzione built-in `locals()` restituisce il namespace corrente sotto forma di dizionario, per cui, se chiamata all'interno di una funzione, di un metodo, di una classe o di una comprehension, restituisce il corrispondente namespace locale. Vediamo i vari casi, iniziando dal namespace locale a una funzione:

```
>>> def foo(par1, par2):
...     a = 33
...     print(locals())
...
>>> foo('ciao', [1, 2, 3])
{'a': 33, 'par2': [1, 2, 3], 'par1': 'ciao'}
>>> f = lambda x, y: locals()
>>> f('python', 100)
{'x': 'python', 'y': 100}
```

Ecco cosa accade se viene chiamata in una classe:

```
>>> class Foo():
...     a = 33
...     def method(self):
...         a = 99
...         b = 100
...         self.c = 200
...         print(locals())
...         [print(k, v) for k, v in list(locals().items()) if not k.startswith('__')]
...
method <function Foo.method at 0xb724489c>
a 33
>>> foo = Foo()
>>> foo.method()
{'a': 99, 'b': 100, 'self': <__main__.Foo object at 0xb724af2c>}
```

Infine in una comprehension:

```
>>> i = 10
>>> [print(locals()) for i in range(3)] # Due etichette locali: i e l'iteratore
```

```

{'i': 0, '.0': <range_iterator object at 0xb7335cc8>}
{'i': 1, '.0': <range_iterator object at 0xb7335cc8>}
{'i': 2, '.0': <range_iterator object at 0xb7335cc8>}
[None, None, None]
>>> i
10
>>> j = 10
>>> {print(locals()) for j in range(3)}
{'j': 0, '.0': <range_iterator object at 0xb7335cf8>}
{'j': 1, '.0': <range_iterator object at 0xb7335cf8>}
{'j': 2, '.0': <range_iterator object at 0xb7335cf8>}
{None}
>>> j
10
>>> i = 10
>>> {i: print(locals()) for i in range(3)}
{'0': <range_iterator object at 0xb74c0d58>, 'i': 0}
{'0': <range_iterator object at 0xb74c0d58>, 'i': 1}
{'0': <range_iterator object at 0xb74c0d58>, 'i': 2}
{0: None, 1: None, 2: None }
>>> i
10

```

Il dizionario restituito da `locals()` viene costruito quando il programma è in esecuzione e quindi rappresenta il namespace locale al momento della chiamata:

```

>>> def foo(par):
...     a = 33
...     print('Prima chiamata a locals(): ', locals())
...     b = 44
...     print('Seconda chiamata a locals(): ', locals())
...
>>> foo('python')
Prima chiamata a locals(): {'par': 'python', 'a': 33}
Seconda chiamata a locals(): {'par': 'python', 'b': 44, 'a': 33}

```

Quando una funzione viene chiamata in modo ricorsivo, a ogni chiamata viene generato un nuovo namespace locale:

```

>>> def recursion(num):
...     if num in range(1, 3):
...         a = 13
...     if num in range(3, 5):
...         b = 35
...     print(locals())
...     if num > 0:

```

```
... recursion(num - 1)
...
>>> recursion(4)
{'b': 35, 'num': 4}
{'b': 35, 'num': 3}
{'a': 13, 'num': 2}
{'a': 13, 'num': 1}
{'num': 0}
```

Come abbiamo detto, `locals()` restituisce il dizionario del namespace corrente, per cui, se viene chiamata al top-level di un modulo, restituisce il dizionario del namespace globale:

```
>>> locals() is globals()
True
```

Un'altra funzione built-in che restituisce il dizionario di un namespace è `vars()`. Se chiamata senza argomenti, restituisce il dizionario del namespace corrente:

```
>>> def foo(par):
...     a = 99
...     print(vars())
...     print(vars() is locals())
...
>>> foo('ciao')
{'par': 'ciao', 'a': 99}
True
>>> vars() is globals() is locals()
True
```

Se le si passa un oggetto `obj`, restituisce il namespace di `obj` come dizionario:

```
>>> def foo():
...     a = 44
...     b = 99
...     print(foo.c)
...
>>> vars(foo) is foo.__dict__
True
>>> vars(foo)['c'] = 33
>>> foo()
33
>>> class Foo:
...     def __init__(self, a):
...         self.a = a
...     def method(self, b, c):
```

```

... self.b = b
... d = 100
...
>>> for k, v in vars(Foo).items():
...     print(k, v)
...
('__module__', '__main__')
('__doc__', None)
('method', <function method at 0xb74165dc>)
('__init__', <function __init__ at 0xb74165a4>)
('d', 100)
>>> f = Foo(44)
>>> vars(f) is f.__dict__
True
>>> vars(f)
{'a': 44}
>>> f.method(55, 66)
>>> vars(f)
{'a': 44, 'b': 55}
>>> f.e = 'python'
>>> vars(f)
{'a': 44, 'b': 55, 'e': 'python'}

```

Scope e risoluzione dei nomi

Quando Python, al momento dell'esecuzione del codice, incontra una etichetta all'interno di una espressione, compie un'operazione detta *risoluzione del nome*, che consiste nell'individuare l'oggetto al quale l'etichetta fa riferimento in quel momento. Le regioni di codice nelle quali Python cerca di risolvere il nome di una etichetta vengono chiamate *scope*. I concetti di namespace e scope sono strettamente legati, poiché uno scope è anche la porzione di codice che genera un namespace.

Python ha quattro scope:

1. *locale*: è lo scope che genera un namespace locale;
2. *enclosing*: è lo scope locale *di una funzione* che contiene al suo interno un altro scope locale;
3. *globale*: è lo scope che genera il namespace globale;
4. *built-in*: è lo scope che genera il namespace built-in.

Risoluzione dei nomi nello scope locale

Una etichetta locale viene risolta con l'oggetto al quale fa riferimento in quel momento nello scope locale. Etichette uguali presenti in altri scope sono, pertanto, trasparenti:

```

>>> a = 33
>>> def foo():

```



```

... a = 99 # Etichetta locale
... print(a) # Viene risolta nello scope locale, poiché è definita in tale scope
...
>>> foo()
99
>>> foo_code = foo.__code__
>>> foo_code.co_varnames # Tupla contenente le variabili locali di `foo`
('a',)

```

Lo scope locale viene determinato durante la fase di compilazione. Quindi ogni etichetta locale viene assegnata staticamente allo scope locale. Questo può essere verificato osservando il bytecode di `foo()`, considerando che l'istruzione `LOAD_FAST(var)` esegue la risoluzione dell'etichetta `var` localmente:

```

>>> import dis
>>> dis.dis(foo)
2 0 LOAD_CONST 1 (99)
3 STORE_FAST 0 (a)
3 6 LOAD_GLOBAL 0 (print)
9 LOAD_FAST 0 (a)
12 CALL_FUNCTION 1 (1 positional, 0 keyword pair)
15 POP_TOP
16 LOAD_CONST 0 (None)
19 RETURN_VALUE

```

Infatti, quando Python incontra una `def`, prima di generare il bytecode annota tutti gli assegnamenti interni alla funzione e per ognuno di questi crea una etichetta locale. Questo comporta che il seguente codice dia luogo a un errore:

```

>>> var = 99 # Etichetta globale
>>> def moo():
...     print(var)
...     var = 44 # Questo assegnamento implica che `var` è locale a `moo()`
...
>>> moo()
Traceback (most recent call last):
...
UnboundLocalError: local variable 'var' referenced before assignment

```

Python, infatti, trova all'interno di `moo()` l'assegnamento `var = 44`, per cui assegna `var` allo scope locale:

```

>>> import dis
>>> dis.dis(moo)

```

```

2 0 LOAD_GLOBAL 0 (print)
3 LOAD_FAST 0 (var)
6 CALL_FUNCTION 1 (1 positional, 0 keyword pair)
9 POP_TOP
3 10 LOAD_CONST 1 (44)
13 STORE_FAST 0 (var)
16 LOAD_CONST 0 (None)
19 RETURN_VALUE

```

Quando, però, la funzione viene chiamata e si arriva all'esecuzione dell'istruzione `print(var)`, poiché Python sa che `var` è una etichetta locale, cerca la sua definizione nelle istruzioni interne alla `def` che precedono la `print()`, ma non la trova.

Se lo scope di una etichetta viene stabilito in fase di compilazione, si dice che la determinazione dello scope è *statica* (*lexical scoping* o *static scoping*). Uno scope definito in modo statico è detto *scope statico* (*lexical scope* o *static scope*).

Se una funzione ha una etichetta locale con lo stesso nome di una globale, dal namespace locale si può accedere all'etichetta globale importando il modulo che contiene il codice:

```

>>> a = 'python'
>>> def foo():
...     a = (1, 2, 3)
...     m = __import__(foo.__module__) # Importo questo modulo
...     print(a)
...     print(m.a)
...     m.a = 100
...
>>> a
'python'
>>> foo()
(1, 2, 3)
python
>>> a
100

```

NOTA

In questo caso avremmo potuto importare all'interno di `foo()` direttamente `__main__`, per poi accedere all'etichetta globale `a` con `__main__.a`. Quest'ultima soluzione, però, non è generale e funziona solo in due casi: quando

eseguiamo il codice in modo interattivo e all'interno dei moduli eseguiti direttamente da linea di comando.

L'istruzione `global` consente di modificare lo scope di una etichetta che dovrebbe essere locale, rendendolo invece globale:

```
>>> def foo():
...     global var # `var` sarà globale
...     print(var)
...     var = 44 # Assegniamo 44 all'etichetta globale
...     print(locals())
...
>>> foo() # L'etichetta `var` deve esistere nel namespace globale
Traceback (most recent call last):
...
NameError: global name 'var' is not defined
>>> var = 33
>>> var
33
>>> foo()
33
{}
>>> var
44
>>> def moo():
...     global mvar
...     mvar = 'python'
...     print(locals())
...
>>> 'mvar' in globals()
False
>>> moo()
{}
>>> 'mvar' in globals()
True
>>> mvar
'python'
```

Etichette locali libere assegnate staticamente allo scope enclosing

Se un'etichetta è utilizzata ma non definita in un certo scope, allora si dice che è *libera* (*free* o *unbound*). Se una etichetta `var` è libera in uno scope locale e tale scope è contenuto all'interno di una funzione che definisce l'etichetta `var`, allora `var` viene assegnata allo scope di tale funzione:

```
>>> def moo():
...     var = 33
```

```

... def foo():
...     print(var)
...     return foo
...
>>> f = moo()
>>> f.__name__
'foo'
>>> f()
33
>>> foo_code = f.__code__
>>> foo_code.co_varnames # Tupla contenente le variabili locali a `foo`
()
>>> foo_code.co_freevars # Tupla contenente le variabili libere in `foo`
('var',)
>>> class Foo:
...     var = 33 # Questa etichetta non viene vista negli scope locali dei metodi
...     def moo():
...         var = 'python'
...         def foo():
...             print(var) # L'etichetta `a` è assegnata allo scope enclosing
...             return foo
...
>>> func_foo = Foo.moo()
>>> func_foo.__name__
'foo'
>>> func_foo()
python
>>> a = 99
>>> def moo():
...     a = 55
...     class Foo:
...         a = 33 # Questa etichetta non viene vista negli scope locali dei metodi
...         def __init__(self):
...             print(a) # L'etichetta `a` è assegnata allo scope enclosing
...             Foo()
...
>>> moo()
55

```

In questo caso la funzione `moo()` è detta *enclosing function* e il suo scope è detto *enclosing*.

Anche lo scope enclosing, come quello locale, è statico. Possiamo verificarlo ancora una volta ispezionando il bytecode, tenendo conto del fatto che l'istruzione `LOAD_DEREF(var)` risolve l'etichetta `var` nello scope enclosing:

```

>>> def moo():

```

```

... var = 33
... def foo():
...     return var
...     return foo
...
>>> f = moo()
>>> f(), f.__name__
(33, 'foo')
>>> import dis
>>> dis.dis(f)
4 0 LOAD_DEREF 0 (var)
3 RETURN_VALUE

```

Gli scope enclosing vengono creati solo all'interno delle funzioni:

```

>>> class Foo:
...     b = 10
...     def __init__(self):
...         print(b)
...
>>> Foo() # Una istruzione `class` non crea uno scope enclosing per i suoi metodi
Traceback (most recent call last):
...
NameError: global name 'b' is not defined

```

L'istruzione `nonlocal` consente di modificare lo scope di una etichetta che dovrebbe essere locale, rendendolo invece enclosing:

```

>>> def moo():
...     var = 10
...     print('var in moo() prima della chiamata a foo(): ', var)
...     def foo():
...         var = 20
...         foo()
...     print('var in moo() dopo la chiamata a foo(): ', var)
...
>>> moo()
var in moo() prima della chiamata a foo(): 10
var in moo() dopo la chiamata a foo(): 10
>>> def moo():
...     var = 10
...     print('var in moo() prima della chiamata a foo(): ', var)
...     def foo():
...         nonlocal var

```

```

... var = 20

... foo()

... print('var in moo() dopo la chiamata a foo(): ', var)

...
>>> moo()
var in moo() prima della chiamata a foo(): 10
var in moo() dopo la chiamata a foo(): 20
>>> def moo():

... var = 10

... print('var in moo() prima della chiamata a Foo(): ', var)

... class Foo:

... def __init__(self):

... nonlocal var

... var = 20

... Foo()

... print('var in moo() dopo la chiamata a Foo(): ', var)

...
>>> moo()
var in moo() prima della chiamata a Foo(): 10
var in moo() dopo la chiamata a Foo(): 20

```

Le etichette `nonlocal` devono necessariamente essere assegnate nello scope enclosing:

```

>>> def moo():
... def foo():
... nonlocal var
... var = 44
... return foo
...
SyntaxError: no binding for nonlocal 'var' found

```

NOTA

Per maggiori informazioni sullo scope enclosing e sull'istruzione `nonlocal`, possiamo consultare, rispettivamente, la PEP-0227 e la PEP-3104.

Una etichetta con scope enclosing, come probabilmente abbiamo già notato dai precedenti esempi, resta legata all'oggetto assegnatole nella funzione enclosing, anche se questa ha terminato la sua esecuzione:

```

>>> def chaining(base):

```

```

... def do_chaining(extension):
...     return base + extension
...     return do_chaining
...
>>> func = chaining('abcd') # La stringa 'abcd' viene assegnata al parametro `base`
>>> func.__name__
'do_chaining'
>>> func('efg')
'abcdefg'

```

Questo comportamento è detto *chiusura di funzione* (*function closure* o *lexical closure*) e riguarda anche le funzioni anonime:

```

>>> def mypow(base):
...     return lambda esponente: base ** esponente
...
>>> b02 = mypow(2)
>>> [b02(i) for i in range(5)]
[1, 2, 4, 8, 16]
>>> b02.__name__
'<lambda>'
>>> c = b02.__code__
>>> c.co_varnames # Etichette locali alla funzione anonima
('esponente',)
>>> c.co_freevars # Etichette libere nella funzione anonima
('base',)
>>> import dis
>>> dis.dis(b02)
2 0 LOAD_DEREF 0 (base)
3 LOAD_FAST 0 (esponente)
6 BINARY_POWER
7 RETURN_VALUE

```

Un esempio significativo è quello delle funzioni anonime che hanno una chiusura sullo scope di una comprehension:

```

>>> func_list = [lambda x: x ** y for y in range(5)]
>>> [f(10) for f in func_list]
[10000, 10000, 10000, 10000, 10000]

```

In questo esempio la list comprehension crea una lista di cinque funzioni anonime, ognuna delle quali ha un'etichetta locale x e una libera y .

Quest'ultima è assegnata allo scope enclosing, ovvero allo scope locale alla list comprehension:

```

>>> f0 = func_list[0]
>>> c = f0.__code__

```

```
>>> c.co_varnames # Etichette locali di `f0`
('x',)
>>> c.co_freevars # Etichette libere in `f0`
('y',)
>>> import dis
>>> dis.dis(f0)
1 0 LOAD_FAST 0 (x)
3 LOAD_DEREF 0 (y)
6 BINARY_POWER
7 RETURN_VALUE
```

Dopo che la lista è stata creata, l'etichetta `y` nello scope enclosing fa riferimento all'ultimo oggetto assegnatole, il numero intero 4. Quando le funzioni vengono chiamate, l'espressione `x ** y` viene valutata: l'etichetta locale `x` fa riferimento all'argomento passato alla funzione, il numero intero 10, mentre l'etichetta libera `y` viene risolta nello scope enclosing con il numero intero 4. Ogni funzione, quindi, restituisce `10 ** 4`.

Il caso è analogo al seguente, costruito con funzioni non anonime:

```
>>> def moo():
...     func_list = []
...     for y in range(5):
...         def foo(x):
...             return x ** y
...         lappend(foo)
...     return func_list
...
>>> [f(10) for f in func_list]
[10000, 10000, 10000, 10000, 10000]
```

Nei due casi appena visti, si dice che si ha un *late binding* tra l'etichetta libera e l'oggetto al quale essa è legato, poiché la risoluzione dell'etichetta libera avviene nel momento in cui la funzione viene chiamata e non in quello in cui viene creata. Quando la risoluzione del nome avviene nel momento della definizione e non in quello della chiamata, si parla di *early binding*.

Se negli esempi precedenti si volesse avere un early binding, si potrebbe utilizzare un parametro di default:

```
>>> func_list = [lambda x, y=y_: x ** y for y_ in range(5)]
>>> [f(10) for f in func_list]
[1, 10, 100, 1000, 10000]
```

In questo caso, infatti, `y` è un parametro delle funzioni anonime, quindi non

più una etichetta libera ma locale, e a ogni iterazione gli viene assegnato il corrispondente valore di `y_`:

```
>>> from inspect import getmembers
>>> for f in func_list:
...     print('Valori di default: ', dict(getmembers(f))['__defaults__'])
...     print('Etichette locali: ', f.__code__.co_varnames)
...     print('Etichette libere: ', f.__code__.co_freevars, end='\n\n')
...
Valori di default per i parametri: (0,)
Etichette locali: ('x', 'y')
Etichette libere: ()
```

```
Valori di default per i parametri: (1,)
Etichette locali: ('x', 'y')
Etichette libere: ()
```

```
Valori di default per i parametri: (2,)
Etichette locali: ('x', 'y')
Etichette libere: ()
```

```
Valori di default per i parametri: (3,)
Etichette locali: ('x', 'y')
Etichette libere: ()
```

```
Valori di default per i parametri: (4,)
Etichette locali: ('x', 'y')
Etichette libere: ()
```

Si può ottenere un early binding anche nel seguente modo:

```
>>> func_list = [(lambda y: lambda x: x ** y)(y_) for y_ in range(5)]
>>> [f(10) for f in func_list]
[1, 10, 100, 1000, 10000]
```

In questo caso a ogni iterazione vengono create due funzioni anonime, una enclosing, il cui parametro `y` è assegnato all'etichetta `y_`, e una annidata che risolve la sua etichetta libera `y` nella lambda enclosing e quindi non più nello scope locale alla list comprehension.

Etichette libere assegnate staticamente allo scope globale

Se una etichetta è libera in una funzione e non è definita nello scope enclosing, allora viene assegnata in modo statico allo scope globale:

```
>>> def moo():
```

```

... def foo():
...     print(a)
...     return foo
...
>>> func_foo = moo()
>>> func_foo.__name__
'foo'
>>> import dis
>>> dis.dis(func_foo)
3 0 LOAD_GLOBAL 0 (print)
3 LOAD_GLOBAL 1 (a)
6 CALL_FUNCTION 1 (1 positional, 0 keyword pair)
9 POP_TOP
10 LOAD_CONST 0 (None)
13 RETURN_VALUE
>>> a = 99
>>> func_foo()
99
>>> class Foo:
...     def foo():
...         print(a)
...
>>> dis.dis(Foo.foo)
3 0 LOAD_GLOBAL 0 (print)
3 LOAD_GLOBAL 1 (a)
6 CALL_FUNCTION 1 (1 positional, 0 keyword pair)
9 POP_TOP
10 LOAD_CONST 0 (None)

```

L'istruzione `class` esegue immediatamente il codice contenuto al suo interno, creando il namespace locale sulla base degli assegnamenti impliciti ed espliciti ed eseguendo ogni altro tipo di istruzione:

```

>>> class Foo:
...     import sys
...     print(sys.platform)
...
linux

```

Per questo motivo, se si utilizza una etichetta libera nel corpo di una classe e questa non è definita nello scope globale e neppure in quello built-in, l'errore viene rilevato immediatamente:

```

>>> class Foo:

```

```

... print(a)
...
Traceback (most recent call last):
...
NameError: name 'a' is not defined
>>> a = 99
>>> class Foo:
... print(a)
...
99

```

Risoluzione delle etichette nello scope built-in

Se una etichetta ha scope globale, ma non è definita globalmente, Python tenta di risolvere il suo nome nello scope built-in:

```

>>> def foo():
... print(sum)
...
>>> import dis
>>> dis.dis(foo)
2 0 LOAD_GLOBAL 0 (print)
3 LOAD_GLOBAL 1 (sum)
6 CALL_FUNCTION 1 (1 positional, 0 keyword pair)
9 POP_TOP
10 LOAD_CONST 0 (None)
13 RETURN_VALUE
>>> 'sum' in globals()
False
>>> foo()
<built-in function sum>

```

Nel caso in cui non riesca a risolvere il nome neppure nello scope built-in, allora lancia una eccezione di tipo `NameError`:

```

>>> sum
<built-in function sum>
>>> del __builtins__.__dict__['sum']
>>> sum
Traceback (most recent call last):
...
NameError: name 'sum' is not defined
>>> def foo():
... print(a)
...
>>> foo()
Traceback (most recent call last):
...
NameError: global name 'a' is not defined

```

Installazione dei package

Abbiamo detto, nei precedenti capitoli, che con gli oggetti built-in e con la libreria standard è possibile essere da subito operativi e in grado di scrivere del codice in svariati ambiti. Nonostante ciò, prima o poi ci capiterà di aver bisogno di estendere le funzionalità della nostra installazione di Python con del software di *terze parti* (*third-party*). Se, ad esempio, fossimo interessati a sviluppare delle applicazioni web, potremmo voler utilizzare il web framework *Django*; se volessimo effettuare delle elaborazioni di immagini, ci potrebbe far comodo usare *PIL* (Python Imaging Library); se lavorassimo in ambito scientifico, probabilmente vorremo utilizzare *matplotlib* per produrre dei grafici, oppure *pygame* se volessimo sviluppare dei giochi o delle applicazioni multimediali, e via dicendo.

Come possiamo immaginare, vi sono varie vie che è possibile intraprendere per installare i moduli e, più in generale, le applicazioni Python.

NOTA

A differenza di una libreria, che tipicamente è composta solamente da package e moduli, in un'applicazione sono presenti anche degli script di avvio e dei file di configurazione. Ad esempio, se consideriamo la shell *IPython* installata nella mia macchina, questa è un'applicazione composta da una serie di moduli e package:

```
$ ls /usr/local/lib/python3.4/site-packages/IPython/  
config display.py external __init__.py lib parallel scripts utils  
core extensions frontend kernel nbformat __pycache__ testing zmq
```

da dei file di configurazione:

```
$ ls ~/.config/ipython/  
profile_default README
```

e uno script di avvio:

```
$ which ipython3  
/usr/local/bin/ipython3
```

Il caso migliore è quello in cui esiste una versione dell'applicazione sotto forma di pacchetto di installazione, preparato in modo da poter essere installato in un particolare sistema operativo. Si pensi, ad esempio, a un file eseguibile (estensione *.msi* o *.exe*) nei sistemi Windows, a un pacchetto con estensione *.deb* nelle distribuzioni Linux basate su Debian, un Apple Disk Image (file con estensione *.dmg*) per il Mac OS X, e via dicendo.

Ad esempio, nel caso di *pygame*, alla pagina <http://www.pygame.org/download.shtml> troviamo le versioni di *pygame* per tutti i principali sistemi operativi, compresi quelli per cellulari. In questo caso l'installazione non presenta alcun problema.

Non sempre, però, è possibile reperire dei pacchetti di installazione, sia perché l'applicazione è stata rilasciata solamente come archivio (*tar.gz*, *zip* ecc.) contenente il codice sorgente, sia magari perché ci interessa avere la versione di sviluppo, o per qualsiasi altro motivo. Nelle sezioni che seguono vedremo come installare questo genere di package e applicazioni.

Python distribution utilities: distutils

La maggior parte delle applicazioni Python viene distribuita usando le *Python distribution utilities*, disponibili mediante il modulo `distutils` della libreria standard. Un'applicazione distribuita mediante Distutils è facilmente riconoscibile, poiché è un archivio (ad esempio, un *.zip* o *.tar.gz*) il cui nome è dato dal nome dell'applicazione seguito dal carattere - e dal numero di versione.

Supponiamo, ad esempio, di voler installare la shell *bpython*, alla quale abbiamo accennato nel [Capitolo 1](#), quando discutevamo delle alternative alla shell interattiva built-in. Possiamo reperire il codice sorgente dal Python Package Index, alla pagina web <https://pypi.python.org/pypi/bpython>.

NOTA

Il Python Package Index, indicato con PyPI, è un repository contenente decine di migliaia di applicazioni scritte in Python. La pagina ufficiale di PyPI è <https://pypi.python.org/pypi>. Il Python Package Index contiene le informazioni sui package ospitati. Ogni package ha un nome e un numero di versione. Ad esempio, l'ultima versione del famoso web framework Django (al momento in cui scrivo), la 1.6.1, è reperibile alla pagina <https://pypi.python.org/pypi/Django/1.6.1>.

Scarichiamo, quindi, *bpython* e spacchettiamo l'archivio:

```
$ wget -q http://bpython-interpreter.org/releases/bpython-0.11.tar.gz
$ tar xzf bpython-0.11.tar.gz
```

Un'applicazione distribuita utilizzando *distutils* contiene uno script di setup chiamato *setup.py*, che deve essere eseguito al fine di installare il package, passandogli l'argomento *install*:

```
$ cd bpython-0.11/
$ sudo python setup.py install
```

Il package, a questo punto, è stato installato nella directory *site-package* della nostra installazione:

```
$ ls -l -d /usr/local/lib/python3.4/site-packages/* | grep "bpython$"
/usr/local/lib/python3.4/site-packages/bpython
```

Proviamo, quindi, ad avviare *bpython*:

```
$ bpython
Traceback (most recent call last):
...
ImportError: No module named 'pygments'
```

Nonostante *bpython* sia stato installato, non può essere eseguito poiché dipende dal package *Pygments*, che non è installato nel nostro sistema. Nel file *setup.py*, infatti, è indicato che deve essere installato anche *pygments*:

```
$ grep "requires" -A 2 setup.py
install_requires = [
'pygments'
],
```

Quindi *distutils* non si occupa di installare le dipendenze. Per poter fare ciò ci occorre installare il package *distribute*. Questo può essere installato in modo standard, dopo aver scaricato il *.tar.gz* da PyPI:

```
$ wget -q https://pypi.python.org/packages/source/d/distribute/distribute\
> 0.6.36.tar.gz
$ tar xzf distribute-0.6.36.tar.gz
$ cd distribute-0.6.36/
$ sudo python setup.py install
```

oppure, più semplicemente, si può scaricare ed eseguire lo script *distribute_setup.py*:

```
$ wget -q http://python-distribute.org/distribute_setup.py
$ sudo python distribute_setup.py
```

A questo punto possiamo installare *bpython* e, grazie a *distribute*, verranno installate anche le dipendenze:

```
$ cd ../bpython-0.11/
$ sudo python setup.py install
```

È stato installato uno script di avvio di *bpython* in */usr/local/bin*:

```
$ which bpython
/usr/local/bin/bpython
```

per cui possiamo avviarlo eseguendo il comando *bpython*:

```
$ bpython
```

In [Figura 4.1](#) è mostrato uno screenshot di *bpython*, dal quale possiamo apprezzare l'utilità del *syntax highlighter*, che facilita notevolmente la lettura del codice.

La [Figura 4.1](#) mostra, inoltre, come i package *distribute*, *bpython* e *Pygments* siano stati installati nella directory *site-packages*.

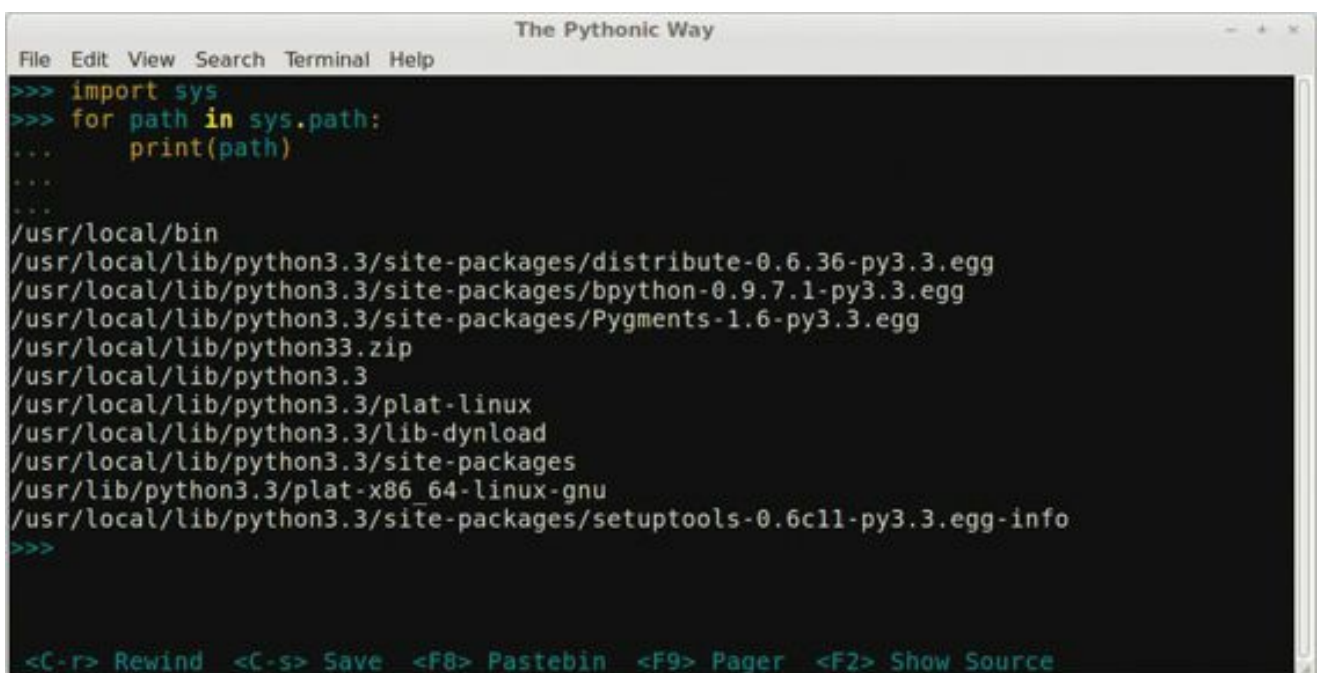


Figura 4.1 - La shell interattiva *bpython* visualizza il codice utilizzando la *syntax highlighter*.

NOTA

Nei sistemi Linux Debian o derivati da Debian (Ubuntu, Mint), quando i package Python o l'interprete Python stesso vengono installati tramite i pacchetti Debian, allora per ospitare le librerie di terze parti viene usata la directory *dist-packages* piuttosto che *site-packages*:

```
$ python2.7 -c "import sys; print('\n'.join(p for p in sys.path[:5] \
> if 'package' in p))" | cut -f1,2 -d'-
/usr/local/lib/python2.7/dist-packages/distribute
/usr/local/lib/python2.7/dist-packages/virtualenv
/usr/local/lib/python2.7/dist-packages/pip
/usr/local/lib/python2.7/dist-packages/bpython
```

Questo non accade, invece, per le installazioni effettuate tramite il codice sorgente. Per maggiori informazioni possiamo consultare la pagina web <http://wiki.debian.org/Python>.

Purtroppo utilizzando *distutils* non è semplice disinstallare le applicazioni, poiché dovremmo rimuovere manualmente i package installati in *site-packages* e gli eventuali script e file di configurazione sparsi nel nostro file system, andando incontro a tutti i rischi che ciò comporta. Nel caso di *bpython*, ad esempio, dovremmo rimuovere i package nella directory di *site-package* e anche lo script di avvio:

```
$ sudo rm /usr/local/lib/python3.4/site-packages/bpython* -r
$ sudo rm /usr/local/bin/bpython*
```

Come possiamo immaginare, questo modo di procedere non è la scelta migliore. La soluzione ai nostri problemi si chiama *pip*.

Pip installs Python

Il modo migliore per installare e gestire i package con Python consiste nell'utilizzare *pip*, acronimo ricorsivo di *pip installs python*. Il modo più semplice per installare *pip* è quello di utilizzare lo script *easy_install*, disponibile grazie a *distribute*:

```
$ sudo easy_install pip
```

oppure possiamo scaricarlo da PyPI per poi installarlo manualmente:


```
$ wget -q https://pypi.python.org/packages/source/p/pip/pip-1.3.1.tar.gz
$ tar xzf pip-1.3.1.tar.gz
$ cd pip-1.3.1/
$ sudo python setup.py install
```

NOTA

Probabilmente abbiamo già sentito parlare di *setuptools* e *easy_install*. Sono i predecessori, rispettivamente, di *distribute* e *pip*. In particolare, *distribute* è un fork di *setuptools*, per cui, quando si installa *distribute*, viene installato anche *setuptools*, mentre *easy_install* è uno script che fa parte del package *setuptools* e che svolge un compito analogo a quello di *pip*. Quindi, riassumendo, quando si installa *distribute* vengono installati anche *setuptools* e *easy_install*. Se vogliamo vedere quali sono i vantaggi di *pip* rispetto a *easy_install*, possiamo consultare la pagina web <http://www.pip-installer.org/en/latest/other-tools.html>. Nella documentazione ufficiale, alla pagina <http://docs.python.org/3/install/>, è raccomandato l'utilizzo di *pip* per l'installazione dei package Python. Per maggiori dettagli si può consultare la PEP-0453.

Anche questa volta è stato installato sia il package nella directory *site-packages*:

```
$ ls -l -d /usr/local/lib/python3.4/site-packages/pip*
/usr/local/lib/python3.4/site-packages/pip-1.3.1-py3.4.egg
```

sia uno script di avvio:

```
$ which pip
/usr/local/bin/pip
```

NOTA

Come possiamo vedere, nella directory *site-packages* non è presente il package *pip*, ma piuttosto la directory *pip-1.3.1-py3.4.egg*. I Python *egg* sono delle directory che vengono aggiunte ai percorsi di ricerca di Python, contenenti al loro interno sia il package sia alcuni metadati utilizzati per facilitare la distribuzione del package stesso. Nel caso di *pip*, ad esempio, il Python *egg* contiene il package *pip* e la directory *EGG-INFO*:

```
$ ls /usr/local/lib/python3.4/site-packages/pip-1.3.1-py3.4.egg
EGG-INFO pip
```

La directory *EGG-INFO* è composta una serie di file contenenti dei metadati:

```
$ tree /usr/local/lib/python3.4/site-packages/pip-1.3.1-py3.4.egg/EGG-INFO/
/usr/local/lib/python3.4/site-packages/pip-1.3.1-py3.4.egg/EGG-INFO/
├── dependency_links.txt
├── entry_points.txt
├── not-zip-safe
├── PKG-INFO
├── requires.txt
├── SOURCES.txt
└── top_level.txt
```

```
0 directories, 7 files
```

Per importare un package non si utilizza il nome del corrispondente *egg*, ma semplicemente il nome del package. Infatti, quando viene installato un Python egg, viene aggiornato un file con suffisso *.pth* presente in *site-packages*, contenente il percorso del package:

```
$ head /usr/local/lib/python3.4/site-packages/easy-install.pth | grep pip
./pip-1.3.1-py3.4.egg
```

In questo modo la directory */usr/local/lib/python3.4/site-packages/pip-1.3.1-py3.4.egg* fa parte dei percorsi di ricerca:

```
$ python -c "import sys; print('\n'.join(p for p in sys.path if 'pip' in p))"
/usr/local/lib/python3.4/site-packages/pip-1.3.1-py3.4.egg
```

e quindi siamo in grado di importare *pip*:

```
$ python -c "import pip; print(pip)"
<module 'pip' from '/usr/local/lib/python3.4/site-packages/pip-1.3.1-py3.4.egg/pip/__init__.py'>
```

Per poter installare un package e le sue dipendenze da PyPI si dà il comando `pip install nome_package`. Ad esempio, per installare *bpython*:

```
$ sudo pip install bpython
```

È possibile scegliere anche una specifica versione:

```
$ sudo pip install bpython==0.11 # Specifica versione
```

o una versione minima:

```
$ sudo pip install bpython>=0.11 # Versione minima
```

Per aggiornare un package con l'ultima versione disponibile su PyPI si usa lo switch `--upgrade`:

```
$ sudo pip install --upgrade bpython
```

NOTA

Negli esempi precedenti abbiamo utilizzato il comando `sudo` (*superuser do*) poiché tipicamente l'utente normale non ha accesso in scrittura alla directory *site-packages* dell'installazione di Python. Spesso questo può essere un problema, che, come vedremo tra breve, viene risolto dall'utilizzo degli ambienti virtuali. Nel seguito, per semplicità, ometteremo il comando `sudo`.

Per indicare a *pip* di installare un package di nome `package_name` a partire da un sistema di controllo di versione del software, si usa lo switch `-e` (*editable installs*):

```
$ pip install -e git+https://git.repo/some_pkg.git#egg=package_name # Da git
$ pip install -e git+https://git.repo/some_pkg.git@foo#egg=package_name # Dal branch 'foo'
$ pip install -e hg+https://hg.repo/some_pkg.git#egg=package_name # Da mercurial
$ pip install -e svn+svn://svn.repo/some_pkg/trunk/#egg=package_name # Da svn
```

Se vogliamo disinstallare un package, diamo il comando `pip uninstall nome_package`. Ad esempio, nel caso di *bpython* diamo il comando `pip uninstall bpython`. Se vogliamo disinstallare anche alcune dipendenze, possiamo elencarle in un file che poi passiamo da linea di comando a *pip* usando l'opzione `-r`. Ad esempio, se oltre a *bpython* vogliamo disinstallare anche *pygments*, procediamo come mostrato di seguito:

```
$ echo "pygments" > reqfile # Creo un file di nome 'reqfile'
$ head reqfile
pygments
$ pip uninstall bpython -r reqfile
```

Come abbiamo visto, *pip* installa anche le dipendenze. In ogni caso, è possibile indicarle manualmente, così come è stato fatto nel caso della disinstallazione:

```
$ pip install bpython -r reqfile
```

Oltre ai comandi `install` e `uninstall`, *pip* supporta una serie di altri comandi, sui quali possiamo ottenere informazioni utilizzando lo switch `-h`:

```
$ pip -h
```

Usage:
`pip <command> [options]`

Commands:

`install` Install packages.

`uninstall` Uninstall packages.

`freeze` Output installed packages in requirements format.

`list` List installed packages.

`show` Show information about installed packages.

`search` Search PyPI for packages.

`zip` Zip individual packages.

`unzip` Unzip individual packages.

`bundle` Create pybundles.

`help` Show help for commands.

General Options:

`-h, --help` Show help.

`-v, --verbose` Give more output. Option is additive, and can be used up to 3 times.

`-V, --version` Show version and exit.

`-q, --quiet` Give less output.

`--log <file>` Log file where a complete (maximum verbosity) record will be kept.

`--proxy <proxy>` Specify a proxy in the form `[user:passwd@]proxy.server:port`.

`--timeout <sec>` Set the socket timeout (default 15 seconds).

`--exists-action <action>` Default action when a path already exists: (s)witch, (i)gnore, (w)ipe, (b)ackup.

`--cert <path>` Path to alternate CA bundle.

Ne esamineremo alcuni nelle prossime sezioni. È possibile utilizzare lo switch `-h` anche per ottenere un help di uno specifico comando:

```
$ pip show -h # Mostra un help del comando 'show'
```

Usage:
`pip show [options] <package> ...`

Description:

Show information about one or more installed packages.

Show Options:

-f, --files Show the full list of installed files for each package.

General Options:

-h, --help Show help.

-v, --verbose Give more output. Option is additive, and can be used up to 3 times.

-V, --version Show version and exit.

-q, --quiet Give less output.

--log <file> Log file where a complete (maximum verbosity) record will be kept.

--proxy <proxy> Specify a proxy in the form [user:passwd@]proxy.server:port.

--timeout <sec> Set the socket timeout (default 15 seconds).

--exists-action <action> Default action when a path already exists: (s)witch, (i)gnore, (w)ipe, (b)ackup.

--cert <path> Path to alternate CA bundle.

Nella sezione *Ambienti virtuali* parleremo ancora di *pip* e vedremo come utilizzarlo in combinazione con *pyvenv* o con *Virtualenv*.

Ambienti virtuali

Un *ambiente virtuale* è una installazione Python nella quale le librerie e gli script sono isolati rispetto alle altre installazioni, ovvero un ambiente Python nel quale i percorsi di ricerca (per default) e gli script sono differenti da quelli delle altre installazioni.

Vediamo di chiarire meglio il concetto. Supponiamo di aver installato tre differenti versioni di Python:

```
$ which python2.7
/usr/bin/python2.7
$ which python3.2
/usr/bin/python3.2
$ which python3.4
/usr/bin/python3.4
```

Queste tre installazioni sono isolate, nel senso che a ciascuna sono associati i propri script e ciascuna contiene la propria directory delle librerie di terze parti:

```
$ python2.7 -c "import sys; print('\n'.join(path for path in sys.path[:4] \
> if 'packages' in path))"
/usr/local/lib/python2.7/dist-packages/distribute-0.6.35-py2.7.egg
/usr/local/lib/python2.7/dist-packages/virtualenv-1.9.1-py2.7.egg
/usr/local/lib/python2.7/dist-packages/pip-1.3.1-py2.7.egg
$ python3.2 -c "import sys; print('\n'.join(path for path in sys.path[:4] \
> if 'packages' in path))"
/usr/local/lib/python3.2/dist-packages/distribute-0.6.36-py3.2.egg
/usr/local/lib/python3.2/dist-packages/pip-1.3.1-py3.2.egg
$ python3.4 -c "import sys; print('\n'.join(path for path in sys.path[:4] \
> if 'packages' in path))"
/usr/local/lib/python3.4/site-packages/distribute-0.6.36-py3.4.egg
/usr/local/lib/python3.4/site-packages/pip-1.3.1-py3.4.egg
```

Python 2.7 è la versione utilizzata dal sistema operativo (Linux Mint) e per questo motivo viene detta *installazione di sistema*. Poiché la distribuzione Linux Mint è derivata da Ubuntu (quindi da Debian), Python 2.7 installa le librerie di terze parti nella propria directory *dist-packages*.

Python 3.2 è stato installato tramite pacchetto Debian, per cui anch'esso installa le librerie nella propria *dist-packages*. Python 3.4 è stato invece installato dai sorgenti, per cui installa le librerie nella sua directory *site-packages*.

Tutte e tre le installazioni sono dette *globali*, nel senso che sono accessibili in lettura da tutti gli utenti. Per ciascuna versione di Python è stato installato *pip*:

```
$ which pip-2.7 # pip relativo a Python 2.7
/usr/local/bin/pip-2.7
$ which pip-3.2 # Pip relativo a Python 3.2
/usr/local/bin/pip-3.2
$ which pip-3.4 # Pip relativo a Python 3.4
/usr/local/bin/pip-3.4
```

Quando eseguiamo `sudo pip-2.7 install some_package` viene installato il package `some_package` nella directory *dist-packages* di Python 2.7, mentre quando eseguiamo `pip-3.2` e `pip-3.4` i package vengono installati, rispettivamente, nella directory *dist-packages* di Python 3.2 e *site-packages* di Python 3.4.

Utilizzando questa configurazione, nella quale le installazioni di Python sono *globali*, si va incontro a diversi problemi. Innanzitutto, un utente che non può effettuare operazioni da *superuser* tipicamente non potrà installare i package in *dist-packages* o *site-packages*. Per questo motivo, se ricordate, nella sezione *Pip installs Python* abbiamo utilizzato il comando `sudo` per installare i package.

Un altro problema sorge quando si ha la necessità di installare differenti versioni dello stesso package. Supponiamo, infatti, di avere un'applicazione *myapp-1* che fa uso del package *mypack-1* e che si voglia installare un'applicazione *myapp-2* che fa uso della versione 2 di *mypack*. Se installiamo anche *mypack-2*, potrebbe accadere, ad esempio, che *mypack* venga aggiornata alla versione 2 e che *myapp-1*, che dipendeva da *mypack-1*, non funzioni più. Per questo motivo l'aggiornamento e la disinstallazione dei package nelle installazioni globali dovrebbero essere fatte con cognizione di causa: se a un'applicazione alla quale hanno accesso tutti gli utenti (*applicazione globale*) vengono a mancare delle dipendenze, si crea un malfunzionamento globale del sistema, o, peggio ancora, possono manifestarsi problemi di sicurezza.

L'ideale, quindi, sarebbe poter creare degli ambienti Python isolati, nei quali l'*utente normale* (non *superuser*) possa installare i package in modo sicuro, ovvero senza correre il rischio di aggiornare o rimuovere le dipendenze di applicazioni Python globali. In questo modo potremmo installare tutti i package che vogliamo e anche diverse versioni dello stesso package in ambienti Python differenti.

A partire da Python 3.3 (PEP-0405) è possibile creare questo genere di installazioni, dette *ambienti virtuali*, mediante il modulo `venv` della libreria

standard.

NOTA

Prima di Python 3.3 gli ambienti virtuali venivano creati utilizzando prevalentemente un tool di terze parti chiamato *Virtualenv*, reperibile da PyPI o dal sito web <http://www.virtualenv.org/>.

Il modulo `venv` della libreria standard

Come abbiamo appena detto, a partire da Python 3.3 possiamo creare un ambiente virtuale usando il modulo `venv` della libreria standard. Per fare ciò si può eseguire direttamente il modulo, oppure, più semplicemente, si può utilizzare lo script *pyvenv*, che viene installato insieme a Python:

```
$ which pyvenv
/usr/local/bin/pyvenv
```

Per creare un ambiente virtuale si passa a *pyvenv* il nome che vogliamo dare all'ambiente:

```
$ pyvenv myvenv
```

NOTA

Nei sistemi Microsoft Windows lo script *pyvenv.py* si trova in *C:\Python34\Tools\Scripts*.

Se non specifichiamo un percorso, l'ambiente virtuale viene creato nella directory di lavoro, in questo caso nella home-directory dell'autore:

```
$ readlink -f myvenv/ # Mostra il percorso completo
/home/marco/myvenv
```

A differenza di ciò che accadeva per le installazioni globali, abbiamo i permessi anche in scrittura:

```
$ ls -ld myvenv/
drwxr-xr-x 5 marco marco 4096 Apr 22 12:32 myvenv/
```


È stata creata la seguente struttura di directory:

```
$ tree myvenv/
myvenv/
├── bin
│   ├── activate
│   ├── pydoc
│   ├── python -> python3.4
│   ├── python3 -> python3.4
│   └── python3.4 -> /usr/local/bin/python3.4
├── include
├── lib
│   └── python3.4
├── site-packages
└── pyvenv.cfg
```

5 directories, 6 files

Nella directory *bin* sono presenti tre link simbolici (*python*, *python3* e *python3.4*) che puntano a */usr/local/bin/python3.4*, ovvero alla versione di Python utilizzata per creare l'ambiente virtuale. Per poter eseguire Python nell'ambiente virtuale dobbiamo *attivare* l'ambiente, eseguendo lo script *bin/activate*:

```
$ which python
/usr/bin/python
$ source myvenv/bin/activate # Attiviamo l'ambiente virtuale `myvenv`
(myvenv) $ which python
/home/marco/myvenv/bin/python
```

Come possiamo vedere, quando l'ambiente virtuale è attivo, il prompt è preceduto dal nome dell'ambiente, posto tra parentesi. Inoltre `sys.prefix` e `sys.exec_prefix` puntano alla directory base dell'ambiente virtuale, mentre `sys.base_prefix` e `sys.base_exec_prefix` puntano all'installazione usata per creare l'ambiente:

```
(myvenv) $ python -c "import sys; print(sys.prefix), print(sys.exec_prefix)"
/home/marco/myvenv
/home/marco/myvenv
(myvenv) $ python -c "import sys; print(sys.base_prefix), print(sys.base_exec_prefix)"
/usr/local
/usr/local
```

Il file *pyvenv.cfg* contiene la configurazione dell'ambiente virtuale:

```
(myvenv) $ cat myvenv/pyvenv.cfg
home = /usr/local/bin
```

```
include-system-site-packages = false
version = 3.4.0
```

L'etichetta `home` rappresenta il percorso della directory contenente la versione di Python utilizzata per creare l'ambiente.

Per default si ha `include-system-site-packages = false` e questa impostazione indica che la directory globale *site-packages* non è inclusa nei percorsi di ricerca:

```
(myvenv) $ python -c "import sys; print('\n'.join(path for path in sys.path \
> if 'package' in path))"
/home/marco/myvenv/lib/python3.4/site-packages
```

Se assegniamo `include-system-site-packages = true`, allora la directory *site-packages* dell'installazione globale verrà inclusa nei percorsi di ricerca:

```
(myvenv) $ sed -i 's/false/true/g' myvenv/pyvenv.cfg # Sostituisco `false` con `true`
(myvenv) $ cat myvenv/pyvenv.cfg
home = /usr/local/bin
include-system-site-packages = true
version = 3.4.0
(myvenv) $ python -c "import sys; print('\n'.join(path for path in sys.path \
> if 'package' in path))"
/usr/local/lib/python3.4/site-packages/distribute-0.6.36-py3.4.egg
/usr/local/lib/python3.4/site-packages/pip-1.3.1-py3.4.egg
/usr/local/lib/python3.4/site-packages/bpython-0.11-py3.4.egg
/home/marco/myvenv/lib/python3.4/site-packages
/usr/local/lib/python3.4/site-packages
```

NOTA

Il fatto che abbiamo indicato `include-system-site-packages = true` non significa che i package verranno installati nella directory *site-packages* globale, ma solamente che le librerie di terze parti installate globalmente saranno disponibili anche per il nostro ambiente virtuale.

Il nostro ambiente virtuale attualmente non contiene alcuna libreria di terze parti:

```
(myvenv) $ tree myvenv/lib/python3.4/site-packages/
myvenv/lib/python3.4/site-packages/
```

```
0 directories, 0 files
```

Procediamo, quindi, con l'installazione di *distribute* e *pip* nell'ambiente

virtuale, per poi installare i package che ci occorrono tramite *pip*. Iniziamo con l'installare *distribute*:

```
(myvenv) $ wget -q http://python-distribute.org/distribute_setup.py
(myvenv) $ python distribute_setup.py
```

Come possiamo vedere, *distribute* è stato installato nella directory *site-packages* dell'ambiente virtuale:

```
(myvenv) $ ls -l -d myvenv/lib/python3.4/site-packages/*
myvenv/lib/python3.4/site-packages/distribute-0.6.36-py3.4.egg
myvenv/lib/python3.4/site-packages/easy-install.pth
myvenv/lib/python3.4/site-packages/setuptools-0.6c11-py3.4.egg-info
myvenv/lib/python3.4/site-packages/setuptools.pth
```

Installiamo anche *pip* tramite *easy_install*:

```
(myvenv) $ easy_install -q pip
(myvenv) $ which pip
/home/marco/myvenv/bin/pip
```

A questo punto il nostro ambiente virtuale è pronto e possiamo installare i package tramite *pip*. Se ricordate, in *pyvenv.cfg* abbiamo assegnato `include-system-site-packages = true` e abbiamo detto che in questo modo la directory *site-packages* dell'installazione globale viene inclusa nei percorsi di ricerca. Infatti, nonostante *bpython* non sia installato in *myvenv*, è installato globalmente, per cui, se proviamo a installarlo, ci viene detto che il package esiste già:

```
(myvenv) $ pip install bpython
Requirement already satisfied (use --upgrade to upgrade): ...
```

Poniamo, quindi, `include-system-site-packages = false` e installiamo *bpython* in *myvenv*:

```
(myvenv) $ sed -i 's/true/false/g' myvenv/pyvenv.cfg
(myvenv) $ pip install -q bpython
(myvenv) $ which bpython
/home/marco/myvenv/bin/bpython
```

Per disattivare l'ambiente virtuale diamo il comando `deactivate`:

```
(myvenv) $ deactivate # Disattiviamo l'ambiente virtuale `myvenv`
$ which python
/usr/bin/python
```

Replica degli ambienti

Uno dei punti di forza di *pip* è che ci consente di creare delle repliche di una data installazione, ovvero di creare una installazione Python avente una libreria di terze parti identica a quella di un'altra installazione Python. Creiamo, ad esempio, un ambiente virtuale *myvenv1*:

```
$ pyvenv myvenv1
```

installiamo *distribute* e *pip*:

```
$ wget -q http://python-distribute.org/distribute_setup.py
$ source myvenv1/bin/activate
(myvenv1) $ python distribute_setup.py
(myvenv1) $ easy_install -q pip
```

e anche *Django* e *gevent*:

```
(myvenv1) $ pip install -q django
(myvenv1) $ pip install -q gevent
```

A questo punto *myvenv1* contiene i seguenti package:

```
(myvenv1) $ pip list
bpython (0.12)
distribute (0.6.36)
Django (1.5.1)
Pygments (1.6)
```

Se vogliamo replicare questo ambiente, dobbiamo creare un file contenente l'elenco dei package presenti nella nostra installazione. Utilizzeremo questo file in un secondo momento per replicare l'installazione stessa.

Per ottenere uno snapshot dell'installazione si utilizza il comando `pip freeze`:

```
(myvenv1) $ pip freeze
Django==1.5.1
Pygments==1.6
bpython==0.12
distribute==0.6.36
```

Salviamo, quindi, questo snapshot reindirizzando l'output di `pip freeze` su file:

```
(myvenv1) $ pip freeze > requirements
(myvenv1) $ deactivate # Disattiviamo l'ambiente virtuale
```

Adesso creiamo un secondo ambiente virtuale e installiamo *distribute* e *pip*:

```
$ pyvenv myenv2
$ source myenv2/bin/activate
(myenv2) $ python distribute_setup.py
(myenv2) $ easy_install -q pip
```

In questo nuovo ambiente è installato, oltre a *pip*, solo *distribute*:

```
(myenv2) $ pip list
distribute (0.6.36)
```

Per replicare *myenv1* su *myenv2* usiamo il comando `pip install` in combinazione con lo switch `-r`, passando come argomento il file *requirements* precedentemente salvato:

```
(myenv2) $ pip install -r requirements
```

Come possiamo vedere, adesso *myenv2* contiene gli stessi package di *myenv1*:

```
(myenv2) $ pip list
bpython (0.12)
distribute (0.6.36)
Django (1.5.1)
Pygments (1.6)
```

Esercizio conclusivo

Concludiamo la prima parte del libro con un interessante esercizio, grazie al quale impareremo a:

- scrivere in modo efficace e tenere aggiornata la documentazione dei nostri moduli e delle nostre funzioni utilizzando una metodologia chiamata *docstring validation testing*;
- *distribuire* i nostri package, e più in generale le nostre applicazioni, in modo tale che anche altri utenti possano installarli e utilizzarli.

Rispetto ai precedenti esercizi conclusivi, non avremo quindi a che fare con un semplice script, ma con un'applicazione. Questa è composta da quattro file e una directory:

\$ tree

```
.
├── pyfinder.py
├── README
├── scripts
│   └── pyfinder
└── setup.py
```

1 directory, 4 files

La prima parte dell'esercizio sarà dedicata all'apprendimento del cosiddetto *docstring validation testing*, mentre nella seconda parte ci occuperemo delle procedure da seguire per distribuire le applicazioni.

Il *docstring validation testing*

In questa sezione parleremo di una particolare modalità di test, detta *docstring validation testing* (test di validazione delle docstring) e indicata con DVT, che si basa sull'utilizzo del modulo `doctest` della libreria standard.

Introduzione al modulo `doctest` della libreria standard

Il modulo `doctest` è noto ai pythonisti sin dal lontano 6 marzo 1999, quando Tim Peters lo pubblicò sul gruppo di discussione `comp.lang.python`, allegato a un messaggio dal titolo *docstring-driven testing*.

Nel suo messaggio Tim fece precedere gli esempi di utilizzo del modulo da

alcune considerazioni, tra le quali le seguenti:

1. Gli esempi non hanno prezzo.
2. Gli esempi che non funzionano sono peggio di quelli inutili.
3. Gli esempi che funzionano alla fine diventano esempi che non funzionano.
4. Le docstring troppo spesso non vengono scritte.
5. Quando le docstring vengono scritte, raramente contengono esempi di grande valore.
6. Le rare docstring che contengono esempi di grande valore alla fine diventano docstring i cui esempi non funzionano.

Il primo punto ci dice che *gli esempi non hanno prezzo*, e questo lo si capisce immediatamente se si confronta la docstring della seguente funzione:

```
def solver(eq: str, var='x'):
    """Restituisci la soluzione di una equazione lineare di primo grado in una incognita."""
    c = eval(eq.replace('=', '-(') + ')', {var: 1j})
    return -c.real / c.imag
```

con la docstring della medesima funzione, ma questa volta provvista di qualche esempio di utilizzo:

```
$ cat solver1.py
def solver(eq: str, var='x'):
    """Restituisci la soluzione di una equazione lineare di primo grado in una incognita.
```

La funzione `solver()` prende come primo argomento una stringa rappresentativa di una equazione di primo grado in una incognita e restituisce la soluzione dell'equazione:

```
>>> solver('2*x + 1 = x')
-1.0
```

Il secondo argomento, che per default vale 'x', consente di specificare la variabile:

```
>>> solver('2*y + 1 = y', var='y')
-1.0
```

Nel caso in cui l'equazione non ammetta una unica soluzione, la funzione solleva una eccezione:

```
>>> solver('x = x')
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
"""
c = eval(eq.replace('=', '-(') + ')', {var: 1j})
return -c.real / c.imag
```

Una docstring contenente degli esempi ha quindi molto più valore di una docstring che si limita semplicemente a descrivere a parole il significato del codice.

NOTA

Nella funzione `solver()`, la chiamata al metodo `eq.replace()` sostanzialmente riporta il secondo membro dell'equazione al primo membro. Dopodiché questa nuova equazione viene passata a `eval()` come primo argomento e `eval()` la valuta sostituendo la variabile con l'unità immaginaria $1j$. Il risultato è, quindi, un numero complesso assegnato all'etichetta `c`. La parte immaginaria di `c` rappresenta il termine a di una equazione nella forma $a \cdot x + b = 0$, mentre la parte reale di `c` rappresenta il termine b , per cui la soluzione $-b/a$ è data da $-c.real / c.imag$. Questa funzione è stata proposta da Maxim Krikun nella ricetta `ActiveState` dal titolo *Linear equations solver in 3 lines*, che possiamo trovare alla pagina <http://code.activestate.com/recipes/365013/>.

La seconda considerazione dice giustamente che *gli esempi che non funzionano sono peggio di quelli inutili*. Probabilmente tutti noi sappiamo quanto tempo si può perdere a cercare di capire degli esempi errati; tipicamente, quando non vi è alcun esempio, si dedica meno tempo a capire il significato del codice di quello che si impiegherebbe se si cercasse di capirlo sulla base di un esempio errato. Morale della favola: se gli esempi funzionano, allora non hanno prezzo, altrimenti è meglio non averne.

I punti 4 e 5 sono chiari, per cui non ci soffermeremo a discuterne, come faremo, invece, per il punto 6. Tim dice che *le rare docstring che contengono esempi di grande valore alla fine diventano docstring i cui esempi non funzionano*. Questo purtroppo capita di frequente, perché, mentre si apportano delle modifiche al codice, tipicamente non si aggiorna immediatamente la documentazione, ma si rinvia questa attività al momento in cui il codice sarà funzionante. Però, una volta completato l'aggiornamento del codice, spesso rinviando ancora l'aggiornamento delle docstring perché abbiamo altro codice da implementare o aggiornare e diamo priorità a quest'ultimo. Oppure, visto che spesso alla documentazione non diamo l'importanza che merita, aggiorniamo in modo veloce le docstring senza verificare gli esempi, non accorgendoci, in questo modo, di eventuali errori. Tutto ciò comporta che le docstring che contenevano esempi di grande valore

spesso diventano docstring i cui esempi non funzionano.

Il *docstring validation testing* ci consente di risolvere tutti questi problemi.

Utilizzo del modulo doctest

Vediamo subito un esempio che ci permetta di capire come utilizzare il modulo `doctest`. Consideriamo, a tale scopo, il file *solver1.py* che abbiamo visto nella precedente sezione.

Questo definisce una funzione `solver()` che è documentata a dovere, con degli esempi funzionanti. Se adesso pensiamo all'ultima osservazione di Tim, probabilmente ci viene in mente che sarebbe utilissimo poter verificare in modo automatico la correttezza dei nostri esempi. Il modulo `doctest` ci consente di fare proprio questo. È sufficiente importare il modulo e chiamare la funzione `doctest.testmod()`, come abbiamo fatto nel file *solver2.py*:

```
$ cat solver2.py
def solver(eq: str, var='x'):
    """Restituisci la soluzione di una equazione lineare di primo grado in una incognita.
```

La funzione `solver()` prende come primo argomento una stringa rappresentativa di una equazione di primo grado in una incognita e restituisce la soluzione dell'equazione:

```
>>> solver('2*x + 1 = x')
-1.0
```

Il secondo argomento, che per default vale 'x', consente di specificare la variabile:

```
>>> solver('2*y + 1 = y', var='y')
-1.0
```

Nel caso in cui l'equazione non ammetta una unica soluzione, la funzione lancia una eccezione:

```
>>> solver('x = x')
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
"""
c = eval(eq.replace('=', '-(') + ')', {var: 1j})
return -c.real / c.imag

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Se proviamo a eseguire il modulo con il comando `python solver2.py`, non ci viene

mostrato alcun output. In realtà, però, il modulo `doctest` ha verificato il corretto funzionamento degli esempi presenti nella docstring e non ha mostrato alcun output perché essi sono corretti. Se vogliamo vedere, comunque, l'output, utilizziamo lo switch `-v`:

```
$ python solver2.py -v
Trying:
solver('2*x + 1 = x')
Expecting:
-1.0
ok
Trying:
solver('2*y + 1 = y', var='y')
Expecting:
-1.0
ok
Trying:
solver('x = x')
Expecting:
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
ok
1 items had no tests:
__main__
1 items passed all tests:
3 tests in __main__.solver
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

Come possiamo vedere da questo output, sono stati effettuati tre test, e tutti e tre sono passati. Il modulo `doctest` non fa altro che cercare all'interno delle docstring le porzioni di codice che somigliano a delle sessioni Python interattive. Di ogni porzione vengono eseguite le istruzioni e, nel caso in cui il risultato sia diverso da quello indicato, allora il test fallisce. Ecco, infatti, cosa accade se modifichiamo la funzione in modo che restituisca una stringa:

```
$ cat solver3.py
def solver(eq: str, var='x'):
    """Restituisci la soluzione di una equazione lineare di primo grado in una incognita.
```

La funzione `solver()` prende come primo argomento una stringa rappresentativa di una equazione di primo grado in una incognita e restituisce la soluzione dell'equazione:

```
>>> solver('2*x + 1 = x')
-1.0
```

Il secondo argomento, che per default vale `'x'`, consente di specificare la variabile:

```
>>> solver('2*y + 1 = y', var='y')
-1.0
```

Nel caso in cui l'equazione non ammetta una unica soluzione, la funzione lancia una eccezione:

```
>>> solver('x = x')
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
"""
c = eval(eq.replace('=', '-()' + ')', {var: 1j}))
return '%s = %f' % (var, -c.real / c.imag)
```

```
if __name__ == '__main__':
import doctest
doctest.testmod()
```

e poi eseguiamo nuovamente il test:

```
$ python solver3.py
*****
File "solver3.py", line 7, in __main__.solver
Failed example:
solver('2*x + 1 = x')
Expected:
-1.0
Got:
'x = -1.000000'
*****
File "solver3.py", line 12, in __main__.solver
Failed example:
solver('2*y + 1 = y', var='y')
Expected:
-1.0
Got:
'y = -1.000000'
*****
1 items had failures:
2 of 3 in __main__.solver
***Test Failed*** 2 failures.
```

Come possiamo vedere, il primo test è fallito perché sulla base dell'esempio ci si aspettava come risultato il float 1.0, mentre invece si è ottenuta la stringa 'x = -1.000000'. Il secondo test è fallito per le stesse ragioni.

Dobbiamo considerare concluso il lavoro di aggiornamento del modulo solo quando tutti i test vengono superati. Quindi aggiorniamo la documentazione:

```
$ cat solver4.py
def solver(eq: str, var='x'):
    """Restituisci la soluzione di una equazione lineare di primo grado in una incognita.
```

La funzione `solver()` prende come primo argomento una stringa rappresentativa di una equazione di primo grado in una incognita e restituisce la soluzione dell'equazione:

```
>>> solver('2*x + 1 = x')
'x = -1.000000'
```

Il secondo argomento, che per default vale 'x', consente di specificare la variabile:

```
>>> solver('2*y + 1 = y', var='y')
'y = -1.000000'
```

Nel caso in cui l'equazione non ammetta una unica soluzione, la funzione lancia una eccezione:

```
>>> solver('x = x')
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
"""
c = eval(eq.replace('=', '-(') + ')', {var: 1j})
return '%s = %f' %(var, -c.real / c.imag)

if __name__ == '__main__':
import doctest
doctest.testmod()
```

ed eseguiamo nuovamente i test del modulo:

```
$ python solver4.py -v
Trying:
solver('2*x + 1 = x')
Expecting:
'x = -1.000000'
ok
Trying:
solver('2*y + 1 = y', var='y')
Expecting:
'y = -1.000000'
ok
Trying:
solver('x = x')
Expecting:
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
ok
1 items had no tests:
__main__
1 items passed all tests:
3 tests in __main__.solver
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

Tutti i test sono passati, per cui possiamo considerare concluso il lavoro di aggiornamento del modulo.

NOTA

Il modo più semplice per aggiornare o creare gli esempi è quello di effettuare delle prove in modalità interattiva e poi fare un copia/incolla all'interno della docstring. Se come shell interattiva si usa IPython, allora per default il prompt è differente da quello che `doctest` si aspetta, per cui, prima di fare un copia/incolla del codice interattivo nella docstring, è necessario utilizzare la *magic function* `%doctest_mode`:

```
In [1]: %doctest_mode
Exception reporting mode: Plain
Doctest mode is: ON
>>> for i in range(2):
...   print(i)
...
0
1
```

Per quanto riguarda le eccezioni, è buona norma omettere il traceback stack. Ad esempio, nel caso della seguente eccezione:

```
>>> solver('x = x')
Traceback (most recent call last):
File "<input>", line 1, in <module>
File "./solver2.py", line 24, in solver
return -c.real / c.imag
ZeroDivisionError: float division by zero
```

quando riportiamo l'esempio nella docstring scriviamo semplicemente:

```
>>> solver('x = x')
Traceback (most recent call last):
...
ZeroDivisionError: float division by zero
```

I tre punti, come vedremo tra breve, sono chiamati *ellipsis*. Infine, è una buona pratica documentare le docstring con pochi esempi ma significativi e fare il test del modulo con `doctest` ogniqualvolta si apportano delle modifiche.

Il docstring validation testing viene utilizzato principalmente per verificare la correttezza degli esempi presenti nelle docstring e raramente per verificare la correttezza del codice sulla base degli esempi.

Per effettuare il test del codice generalmente si utilizzano altri tool, come, ad esempio, il modulo `unittest` della libreria standard, che vedremo nel [Capitolo 6](#), oppure *nose*. Si utilizzano tool diversi perché gli scopi sono diversi: quando si fanno i test del codice l'obiettivo è quello di verificare il corretto funzionamento del codice in tutti i possibili casi, mentre l'obiettivo del DVT è quello di verificare la correttezza degli esempi presenti nelle docstring. Approfondiremo questi aspetti nel [Capitolo 6](#).

Il file `pyfinder.py`

Ora che sappiamo in cosa consiste il docstring validation testing, diamo uno sguardo al file `pyfinder.py` della nostra applicazione:

```
$ cat pyfinder.py
```

```
"""
```

Questo modulo definisce delle funzioni generatore che consentono di cercare dei file nel file-system e anche del testo all'interno di un file.

Il modulo definisce le seguenti funzioni:

- `file_finder()`: cerca dei file il cui nome verifica un dato pattern. Nel seguente esempio viene cercato un file di nome `'os.py'` all'interno della directory contenente il modulo `'os'` della libreria standard, per cui viene trovato il file `'os.py'`:

```
>>> for file in file_finder('os.py', os.path.dirname(os.__file__)):
...     print(os.path.basename(file)) # Stampa solo il nome, non il percorso completo
...
...
os.py
```

- `file_inspector()`: cerca un dato pattern all'interno di un file di testo. Ad esempio:

```
>>> for match in file_inspector(doctest.__file__, 'Tim Peters'):
...     print(match, end="")
...
...
# Released to the public domain 16-Jan-2001, by Tim Peters (tim@python.org).
"""
```

```
import fnmatch
import os
import re
```

```
def file_finder(pattern: str, top_dir: str=os.curdir, recursive: bool=False):
    """Cerca dei file il cui nome verifica un dato pattern.
```

La ricerca avviene per default all'interno della directory corrente:

```
>>> for file in file_finder('pyfinder.py'):
...     print(os.path.basename(file)) # Stampa solo il nome, non il percorso completo
...
...
pyfinder.py
```

Un secondo argomento opzionale viene utilizzato per indicare la directory di partenza:

```
>>> for file in file_finder('message.py', os.path.dirname(email.__file__)):
...     print(os.path.basename(file))
...
...
message.py
```

Il pattern può contenere anche i caratteri jolly delle shell Unix-like:

```
>>> for file in file_finder('me*age.py', os.path.dirname(email.__file__)):
...     print(os.path.basename(file))
...
...
...
message.py
```

La ricerca può essere fatta anche in modo ricorsivo, passando come terzo argomento `True`. Nel seguente esempio il file `message.py` viene trovato nella directory `email` (primo risultato) e in una sua sotto-directory (secondo risultato):

```
>>> for file in file_finder('me*age.py', os.path.dirname(email.__file__), True):
...     print(os.path.basename(file))
...
...
message.py
message.py
"""
for path, dirs, files in os.walk(top_dir):
    if not recursive:
        dirs.clear() # Svuota la lista delle sotto-directory di `top_dir`
    for name in fnmatch.filter(files, pattern):
        yield os.path.join(path, name)
```

```
def file_inspector(file_name: str, pattern: str):
    """Cerca un dato pattern all'interno di un file di testo e restituisce un generatore.
```

Nel seguente caso, ad esempio, viene cercato il testo `Regular` all'interno del modulo `re` (file `re.py` della libreria standard di Python):

```
>>> for match in file_inspector(re.__file__, 'Regular'):
...     print(match, end="")
...
...
```

```
# Secret Labs' Regular Expression Engine
Regular expressions can contain both special and ordinary characters.
```

Il pattern può essere una espressione regolare:

```
>>> for match in file_inspector(re.__file__, '^Regular'):
...     print(match, end='')
...
...
Regular expressions can contain both special and ordinary characters.
"""

for line in open(file_name):
    if re.search(pattern, line):
        yield line

if __name__ == '__main__':
    import doctest, email
    doctest.testmod()
```

Come possiamo vedere, il file *pyfinder.py* è un modulo che contiene la definizione delle funzioni `file_finder()` e `file_inspector()` che abbiamo visto nell'esercizio conclusivo del Capitolo 3. È composto da un centinaio linee di codice, ma in realtà, se eliminiamo le stringhe di documentazione e l'istruzione `if` finale, il tutto si riduce alla definizione delle due funzioni generatore `file_finder()` e `file_inspector()`, quindi a una decina di linee di codice. Le due funzioni sono esattamente quelle viste nel precedente capitolo, a parte l'aggiunta del valore di default `os.curdir` per il parametro `top_dir` di `file_finder()`:

```
def file_finder(pattern: str, top_dir: str=os.curdir, recursive: bool=False):
```

Le docstring di `file_finder()` e `file_inspector()` contengono degli esempi che mostrano le due funzioni all'opera. Anche la docstring del modulo contiene degli esempi e anche questi vengono verificati da `doctest` quando si esegue il modulo:

```
$ python pyfinder.py -v
Trying:
for file in file_finder('os.py', os.path.dirname(os.__file__)):
    print(os.path.basename(file)) # Stampa solo il nome, non il percorso completo
```

```
Expecting:
os.py
ok
Trying:
for match in file_inspector(doctest.__file__, 'Tim Peters'):
    print(match, end='')

```

Expecting:

Released to the public domain 16-Jan-2001, by Tim Peters (tim@python.org).

```
ok
Trying:
for file in file_finder('pyfinder.py'):
print(os.path.basename(file)) # Stampa solo il nome, non il percorso completo
Expecting:
pyfinder.py
ok
Trying:
for file in file_finder('message.py', os.path.dirname(email.__file__)):
print(os.path.basename(file))
```

```
Expecting:
message.py
ok
Trying:
for file in file_finder('me*age.py', os.path.dirname(email.__file__)):
print(os.path.basename(file))
```

```
Expecting:
message.py
ok
Trying:
for file in file_finder('me*age.py', os.path.dirname(email.__file__), True):
print(os.path.basename(file))
```

```
Expecting:
message.py
message.py
ok
Trying:
for match in file_inspector(re.__file__, 'Regular'):
print(match, end='')
Expecting:
# Secret Labs' Regular Expression Engine
Regular expressions can contain both special and ordinary characters.
ok
Trying:
for match in file_inspector(re.__file__, '^Regular'):
print(match, end='')
Expecting:
Regular expressions can contain both special and ordinary characters.
ok
3 items passed all tests:
2 tests in __main__
4 tests in __main__.file_finder
2 tests in __main__.file_inspector
8 tests in 3 items.
8 passed and 0 failed.
Test passed.
```

Esempi con risultati indipendenti dal sistema operativo e dall'installazione di Python

Gli esempi utilizzati nelle docstring del modulo `pyfinder` dovrebbero essere abbastanza chiari. Consideriamo il primo:

```
>>> for file in file_finder('os.py', os.path.dirname(os.__file__)):
...     print(os.path.basename(file)) # Stampa solo il nome, non il percorso completo
...
...
os.py
```

Il nostro scopo è quello di avere del codice che dia lo stesso risultato su tutti i sistemi operativi. Per fare ciò dobbiamo cercare un file che sia presente nel sistema di ciascun utente che esegue *pyfinder.py*. Visto che eseguiamo il modulo `pyfinder` con Python, il file *os.py* esiste sicuramente nel nostro sistema. In quale directory, però, dobbiamo cercarlo? Il percorso del file, infatti, è diverso a seconda della versione di Python che si utilizza:

```
$ python3.4 -c "import os; print(os.__file__)" # Linux Mint
/usr/local/lib/python3.4/os.py
$ python3.2 -c "import os; print(os.__file__)" # Linux Mint
/usr/lib/python3.2/os.py
```

e anche a seconda del sistema operativo:

```
>python -c "import os; print(os.__file__)" # Microsoft Windows
C:\Python34\lib\os.py
```

Piuttosto che indicare il percorso completo, usiamo l'etichetta `os.__file__`, visto che questa fornisce proprio il percorso del file *os.py*. Dato che `os.path.dirname(file_path)` restituisce la directory di `file_path`:

```
>>> import os
>>> os.path.dirname('/home/marco/foofile.py')
'/home/marco'
```

la directory in cui si trova *os.py* è data da `os.path.dirname(os.__file__)`.

A questo punto sarà già chiaro, probabilmente, il motivo per cui è stato stampato `os.path.basename(file)` piuttosto che `file`. Se, infatti, scriviamo l'esempio come mostrato nel file *wrong_finder.py*:

```
$ cat wrong_finder.py
#!/usr/bin/env python
"""
>>> for file in file_finder('os.py', os.path.dirname(os.__file__)):
...     print(file)
```

```

...
...
/usr/local/lib/python3.4/os.py
"""

import fnmatch
import os

def file_finder(pattern: str, top_dir: str=os.curdir, recursive: bool=False):
    for path, dirs, files in os.walk(top_dir):
        if not recursive:
            dirs.clear() # Svuota la lista delle sotto-directory di `top_dir`
        for name in fnmatch.filter(files, pattern):
            yield os.path.join(path, name)

if __name__ == '__main__':
    import doctest, email
    doctest.testmod()

```

questo funziona solo se il file *os.py* si trova in */usr/local/lib/python3.4/*, quindi non funziona sicuramente sui sistemi Microsoft Windows, ma nemmeno sullo stesso sistema se si utilizzano diverse installazioni di Python:

```

$ /usr/local/bin/python3.4 wrong_finder.py # OK
$ /usr/bin/python3.4 wrong_finder.py
*****
File "wrong_finder.py", line 3, in __main__
Failed example:
for file in file_finder('os.py', os.path.dirname(os.__file__)):
print(file)

Expected:
/usr/local/lib/python3.4/os.py
Got:
/usr/lib/python3.4/os.py
*****
1 items had failures:
1 of 1 in __main__
***Test Failed*** 1 failures.

```

Verifica della correttezza degli esempi contenuti in un file di testo qualunque

Il modulo `doctest` ci consente anche di verificare il corretto funzionamento degli esempi presenti in un file di testo. Consideriamo, ad esempio, il file della nostra applicazione:

```

$ cat README
=====
Installing and using PyFinder
=====

```

Introduction

PyFinder allows you either to find files in the file-system or some text inside files.

It provides a library (the `pyfinder.py` file) and a script called `pyfinder`, that can be executed from the command line.

Supported Python versions

PyFinder requires Python version 3.3 or above.

Installation instructions

PyFinder is only released as a source distribution. Installing by `pip` <http://www.pip-installer.org/> is the simplest and preferred way on all systems::

```
$ pip install pyfinder
```

Otherwise download the source tarball from <http://pypi.python.org/pypi/pyfinder>, uncompress it, enter the `pyfinder-x.y` directory and then run the `install` command::

```
$ python setup.py install
```

How to use PyFinder

To use the `pyfinder` module just import it and call its functions. Here is some examples::

```
>>> import os
>>> import email
>>> import pyfinder
>>> for file in pyfinder.file_finder('message.py', os.path.dirname(email.__file__)):
...     print(os.path.basename(file))
...
...
message.py
>>> for match in pyfinder.file_inspector(pyfinder.__file__, 'def file_*'):
...     print(match, end="")
...
...
def file_finder(pattern: str, top_dir: str=os.curdir, recursive: bool=False):
def file_inspector(file_name: str, pattern: str):
```

For more information about the module usage look at its documentation (`pyfinder.__doc__`).

There is also a `pyfinder` script that allows you to easily find files and text inside files from the command line. For information about the script usage look at the output of `pyfinder -h`.

Come possiamo vedere, nella sezione *How to use PyFinder* sono mostrati due brevi esempi di utilizzo delle funzioni `file_finder()` e `file_inspector()`. Possiamo verificare in due modi che questi esempi funzionino correttamente. Il primo consiste nel creare uno script Python nel quale viene chiamata la funzione `doctest.testfile()`:

```
$ cat test_myfile.py
import doctest
doctest.testfile('README')
```

che, come al solito, va poi eseguito:

```
$ python test_myfile.py -v
Trying:
import os
Expecting nothing
ok
Trying:
import email
Expecting nothing
ok
Trying:
import pyfinder
Expecting nothing
ok
Trying:
for file in pyfinder.file_finder('message.py', os.path.dirname(email.__file__)):
print(os.path.basename(file))
Expecting:
message.py
ok
Trying:
for match in pyfinder.file_inspector(pyfinder.__file__, 'def file_*'):
print(match, end='')
Expecting:
def file_finder(pattern: str, top_dir: str=os.curdir, recursive: bool=False):
def file_inspector(file_name: str, pattern: str):
ok
1 items passed all tests:
5 tests in README
5 tests in 1 items.
5 passed and 0 failed.
Test passed.
```

Il secondo modo, invece, è ancora più semplice, perché ci consente di eseguire gli esempi di *README* senza dover creare alcuno script:

```
$ python -m doctest README
```

NOTA

La possibilità di verificare la correttezza degli esempi di codice contenuti in un file di testo è molto importante quando si scrivono dei libri, dei manuali o della documentazione in generale. Il modulo `doctest`, infatti, è stato di grande aiuto all'autore per verificare la correttezza degli esempi riportati in questo libro: ogni capitolo è stato esportato come file di testo, dopodiché sono stati

verificati gli esempi eseguendo da linea di comando `python -m doctest capitolo.txt`.

Option flags e direttive

Il modulo `doctest` fornisce una serie di opzioni, dette *option flags*, che consentono di controllare vari aspetti dei test. Queste opzioni possono essere specificate mediante dei commenti speciali detti *direttive*. Vediamo subito un esempio, in modo da capire di cosa stiamo parlando. Consideriamo l'esempio contenuto nel file *test_ellipsis*:

```
$ cat test_ellipsis
Esempio::
>>> for i in range(3):
...   print(i)
...
...
0
5
2
```

Il test di questo esempio fallisce, perché il numero 5 non è contenuto in `range(3)`:

```
$ python -m doctest test_ellipsis
*****
File "test_ellipsis", line 3, in test_ellipsis
Failed example:
for i in range(3):
print(i)
Expected:
0
55555
2
Got:
0
1
2
*****
1 items had failures:
1 of 1 in test_ellipsis
***Test Failed*** 1 failures.
```

Adesso modifichiamo il file nel seguente modo:

```
$ cat test_ellipsis
Esempio::
>>> for i in range(3): # doctest: +ELLIPSIS
...   print(i)
...
...
```

```
0
...
2
```

Il commento `# doctest: +ELLIPSIS` è una direttiva che attiva l'option flag `doctest.ELLIPSIS`. Quando viene attivato questo flag, allora una sequenza di tre punti (...) nell'output viene verificata da qualunque stringa. Infatti adesso il test passa:

```
$ python -m doctest test_ellipsis -v
Trying:
for i in range(3): # doctest: +ELLIPSIS
print(i)
Expecting:
0
...
2
ok
1 items passed all tests:
1 tests in test_ellipsis
1 tests in 1 items.
1 passed and 0 failed.
Test passed.
```

La sequenza di tre punti viene detta *ellipsis* e l'abbiamo già utilizzata più volte per omettere il traceback stack delle eccezioni.

Le direttive sono dei commenti che iniziano con il testo `doctest:.` Il testo successivo è dato dal nome dell'option flag (nel nostro caso `ELLIPSIS` indica che si vuole fare riferimento all'opzione `doctest.ELLIPSIS`) preceduto (senza spazi) da un simbolo `+` o `-`. Il simbolo `+` indica che l'opzione deve essere attivata, mentre il simbolo `-` indica che deve essere disattivata.

Vediamo qualche altra opzione. Consideriamo, ad esempio, il seguente file:

```
$ cat test_skip
>>> from math import pi, cos, log10
>>> print(cos(pi))
-1.0
>>> log10(100)
2.0
```

In questo caso verranno effettuati tre test, uno per ciascuna istruzione:

```
$ python -m doctest test_skip -v
Trying:
from math import pi, cos, log10
Expecting nothing
ok
Trying:
print(cos(pi))
```

```
Expecting:
-1.0
ok
Trying:
log10(100)
Expecting:
2.0
ok
1 items passed all tests:
3 tests in test_skip
3 tests in 1 items.
3 passed and 0 failed.
Test passed.
```

Se vogliamo saltare un test, possiamo utilizzare l'opzione `doctest.SKIP`:

```
$ cat test_skip
>>> from math import pi, cos, log10
>>> print(cos(pi)) # doctest: +SKIP
-1.0
>>> log10(100)
2.0
```

Come possiamo vedere, il test dell'istruzione `print(cos(pi))` non viene eseguito:

```
$ python -m doctest test_skip -v
Trying:
from math import pi, cos, log10
Expecting nothing
ok
Trying:
log10(100)
Expecting:
2.0
ok
1 items passed all tests:
2 tests in test_skip
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

Infine, possiamo utilizzare una direttiva all'interno di un commento, ponendola alla fine del commento:

```
$ cat test_skip
>>> from math import pi, cos, log10
>>> print(cos(pi)) # Stampa il coseno di Pi greco # doctest: +SKIP
-1.0
>>> log10(100)
2.0
$ python -m doctest test_skip
$
```


Per maggiori dettagli sul modulo `doctest`, possiamo consultare la documentazione sul sito ufficiale, alla pagina <http://docs.python.org/3/library/doctest.html>.

Installazione dell'applicazione

Abbiamo detto che la nostra applicazione è strutturata nel modo seguente:

```
$ tree
```

```
.
├── pyfinder.py
├── README
├── scripts
│   └── pyfinder
└── setup.py
```

```
1 directory, 4 files
```

Del file *pyfinder.py* sappiamo già tutto, per cui ci resta da indagare sugli altri tre file. Iniziamo dal più importante, lo script *setup.py*:

```
$ cat setup.py
from distutils.core import setup
setup(
    name = 'pyfinder',
    version = '0.2',
    description = 'Look for files and text inside files',
    long_description = open('README').read(),
    py_modules = ['pyfinder'],
    author = 'Marco Buttu',
    author_email = "marco.buttu@gmail.com",
    license = 'BSD',
    url = 'https://pypi.python.org/pypi/pyfinder/',
    keywords = 'python generators distutils',
    scripts = ['scripts/pyfinder'],
    platforms = 'all',
    classifiers = [
        'Intended Audience :: Education',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License',
        'Operating System :: OS Independent',
        'Programming Language :: Python :: 3.3',
        'Topic :: Documentation',
        'Topic :: Education :: Testing',
        'Topic :: Text Processing :: Filters',
        'Topic :: Utilities'
    ],
)
```

Come possiamo vedere, questo script non fa altro che chiamare la funzione

`core.setup()` di `distutils`, la quale fornisce a `distutils` le informazioni sull'applicazione. Ad esempio, il nome dell'applicazione, che in questo caso è `pyfinder`, è assegnato al parametro `name`, la versione dell'applicazione è `0.2`, e così via per i restanti argomenti, il cui significato è abbastanza intuitivo. Ci soffermiamo un attimo solo sui due parametri `py_modules` e `scripts`. Questi rappresentano rispettivamente la lista dei moduli e quella degli script che dovranno essere installati. I primi andranno installati nella directory in cui risiedono le librerie di terze parti, mentre i secondi nella directory in cui si trova l'interprete, che, ad esempio, nel caso della macchina dell'autore, è `/home/marco/myvenv/bin`:

```
>>> import sys
>>> import os
>>> os.path.dirname(sys.executable)
'/home/marco/myvenv/bin'
```

La nostra applicazione ha un solo script, il file *pyfinder* contenuto nella directory *scripts*:

```
$ cat scripts/pyfinder
#!/python
import argparse
import os
from pyfinder import file_finder, file_inspector

parser = argparse.ArgumentParser(description='Look for files and text inside the files')
parser.add_argument('-f', '--file', type=str, required=True, help='File name')
parser.add_argument('-d', '--dir', default=os.curdir, help='Top directory')
parser.add_argument('-p', '--pattern', default='', help='Pattern to look for')
parser.add_argument('-r', '--recursive', action='store_true', help='Recursive')
args = parser.parse_args()

for file in file_finder(args.file, args.dir, args.recursive):
    if args.pattern:
        for line in file_inspector(file, args.pattern):
            print(file, line, sep=' -> ', end='')
    else:
        print(file)
```

Come possiamo vedere, lo script inizia con la linea `#!/python`. Questa linea verrà sostituita da parte di `distutils` con il percorso dell'interprete. Prima di procedere con la distribuzione dell'applicazione verifichiamo che tutto funzioni come dovrebbe, eseguendo l'installazione nel nostro ambiente virtuale:

```
$ source ~/myvenv/bin/activate
(myvenv) $ python setup.py install
running install
```

```
running build
running build_py
creating build
creating build/lib
copying pyfinder.py -> build/lib
running build_scripts
creating build/scripts-3.4
copying and adjusting scripts/pyfinder -> build/scripts-3.4
changing mode of build/scripts-3.4/pyfinder from 644 to 755
running install_lib
copying build/lib/pyfinder.py -> /home/marco/myenv/lib/python3.4/site-packages
byte-compiling /home/marco/myenv/lib/python3.4/site-packages/pyfinder.py to pyfinder.cpython-34.pyc
running install_scripts
copying build/scripts-3.4/pyfinder -> /home/marco/myenv/bin
changing mode of /home/marco/myenv/bin/pyfinder to 755
running install_egg_info
Writing /home/marco/myenv/lib/python3.4/site-packages/pyfinder-0.2-py3.4.egg-info
```

NOTA

Lo script che chiama la funzione `setup()` di `distutils` non deve necessariamente avere il nome `setup.py`, anche se convenzionalmente viene chiamato in questo modo.

Lo script *pyfinder* è stato installato nella directory *bin* dell'installazione di Python:

```
(myenv) $ which pyfinder
/home/marco/myenv/bin/pyfinder
```

La prima riga è stata sostituita in accordo con quanto detto prima:

```
(myenv) $ head ~/myenv/bin/pyfinder -n 1
#!/home/marco/myenv/bin/python
```

Lo script *pyfinder* si trova nei percorsi di ricerca del nostro sistema, per cui possiamo eseguirlo direttamente da linea di comando:

```
(myenv) $ pyfinder -f "passwd" -d /etc/ -p "marco"
/etc/passwd -> marco:x:1000:1000:Marco Buttu, +39 328 5943689, marco.buttu@gmail.com:/home/marco/bin/bash
```

Il modulo *pyfinder* è stato installato nella directory *site-package*:

```
(myenv) $ ls -l ~/myenv/lib/python3.4/site-packages/pyfinder*
/home/marco/myenv/lib/python3.4/site-packages/pyfinder-0.2-py3.4.egg-info
/home/marco/myenv/lib/python3.4/site-packages/pyfinder.py
```

Come possiamo vedere, è stato creato anche un file con suffisso *.egg-info* contenente i metadati dell'applicazione:

```
(myvenv) $ cat ~/myvenv/lib/python3.4/site-packages/pyfinder-0.2-py3.4.egg-info
```

```
Metadata-Version: 1.1
```

```
Name: pyfinder
```

```
Version: 0.2
```

```
Summary: Look for files and text inside files
```

```
Home-page: https://pypi.python.org/pypi/pyfinder/
```

```
Author: Marco Buttu
```

```
Author-email: marco.buttu@gmail.com
```

```
License: BSD
```

```
Description:
```

```
Installing and using PyFinder
```

```
Introduction
```

```
PyFinder allows you either to find files in the file-system or some text  
inside files.
```

```
It provides a library (the 'pyfinder.py' file) and a script called 'pyfinder',  
that can be executed from the command line.
```

```
...
```

```
Classifier: Topic :: Text Processing :: Filters
```

```
Classifier: Topic :: Utilities
```

NOTA

La descrizione dell'applicazione non è altro che il contenuto del file *README*. Infatti nel file *setup.py* avevamo passato `open('README').read()` al parametro `long_description`.

L'applicazione, quindi, è stata installata correttamente. A questo punto non ci resta che impacchettarla e renderla disponibile agli utenti. Vediamo come fare.

Distribuire le applicazioni

Distribuire un'applicazione significa renderla disponibile agli utenti (ad esempio caricandola su PyPI, pubblicandola su un sito web ecc.) affinché possano scaricarla e installarla. Per poter distribuire un'applicazione dobbiamo prima di tutto creare una cosiddetta *distribuzione* (in inglese *distribution*), che può essere un archivio (*ZIP*, *.tar.gz* ecc.) contenente il codice sorgente, e in questo caso è detta *source distribution*, oppure un *installer* (ad esempio, un *.msi* Windows, un pacchetto Debian, un pacchetto RPM ecc.), e in questi casi è detta *built distribution*. Questo processo è descritto nella sezione *Creare una distribuzione dell'applicazione*. Il secondo

step consiste nel rendere la distribuzione disponibile agli utenti; lo vedremo nella sezione *Pubblicare la distribuzione su PyPI*.

Creare una distribuzione dell'applicazione

Vediamo separatamente come creare una source distribution e una built distribution.

Source distribution

Per creare una source distribution dobbiamo eseguire lo script di setup passandogli come argomento `sdist` (la *s* significa proprio *source*). Attualmente nella directory corrente vi sono solo i sorgenti:

```
$ ls -F # Le directory vengono indicate con un carattere slash finale
pyfinder.py README scripts/ setup.py
```

Creiamo, quindi, una source distribution:

```
$ python setup.py sdist
```

Come possiamo vedere, sono stati creati un file *MANIFEST* e una directory *dist*:

```
$ tree
.
├── dist
│   └── pyfinder-0.2.tar.gz
├── MANIFEST
├── pyfinder.py
├── README
├── scripts
│   └── pyfinder
└── setup.py
```

```
2 directories, 6 files
```

Il file *MANIFEST* contiene l'elenco dei file inclusi nella distribuzione:

```
$ cat MANIFEST
# file GENERATED by distutils, do NOT edit
README
pyfinder.py
setup.py
scripts/pyfinder
```

La directory *dist* contiene la distribuzione sotto forma di archivio *.tar.gz*:

```
$ tar ztf dist/pyfinder-0.2.tar.gz
pyfinder-0.2/
pyfinder-0.2/PKG-INFO
pyfinder-0.2/README
pyfinder-0.2/pyfinder.py
pyfinder-0.2/scripts/
pyfinder-0.2/scripts/pyfinder
pyfinder-0.2/setup.py
```

Il nome dell'archivio è stato creato sulla base degli argomenti passati ai parametri `name` e `version` della funzione `setup()`. Nella distribuzione oltre ai sorgenti è presente il file *PKG-INFO* contenente i metadati dell'applicazione:

```
$ tar xzfO dist/pyfinder-0.2.tar.gz pyfinder-0.2/PKG-INFO | head -n 7
Metadata-Version: 1.1
Name: pyfinder
Version: 0.2
Summary: Look for files and text inside files
Home-page: https://pypi.python.org/pypi/pyfinder/
Author: Marco Buttu
Author-email: marco.buttu@gmail.com
```

Per default nei sistemi Unix-like viene creato un archivio *.tar.gz*, mentre su Windows viene creato un file *ZIP*. È comunque possibile specificare il formato tramite lo switch `--formats`:

```
$ python setup.py sdist --formats=zip,bztar
$ ls dist
pyfinder-0.2.tar.bz2 pyfinder-0.2.tar.gz pyfinder-0.2.zip
```

Come possiamo vedere, sono state create le distribuzioni *pyfinder-0.2.zip* e *pyfinder-1.0.bz2*. Oltre a `zip`, `gztar`, `bztar`, si possono usare i formati `ztar` e `tar`, che producono, rispettivamente, un archivio *.tar.Z* e uno *.tar*.

Built distribution

Per creare una built distribution si passa l'argomento `bdist` allo script di setup:

```
$ python setup.py bdist
```

È stato creato un archivio di nome *pyfinder-0.2.linux-i686.tar.gz*:

```
$ ls dist/
pyfinder-0.2.linux-i686.tar.gz pyfinder-0.2.tar.bz2 pyfinder-0.2.tar.gz pyfinder-0.2.zip
```

Questa distribuzione contiene una struttura di directory che è specifica del sistema nella quale è stata creata:

```
$ tar ztf dist/pyfinder-0.2.linux-i686.tar.gz
./
./usr/
./usr/local/
./usr/local/bin/
./usr/local/bin/pyfinder
./usr/local/lib/
./usr/local/lib/python3.4/
./usr/local/lib/python3.4/site-packages/
./usr/local/lib/python3.4/site-packages/__pycache__/
./usr/local/lib/python3.4/site-packages/__pycache__/pyfinder.cpython-34.pyc
./usr/local/lib/python3.4/site-packages/pyfinder.py
./usr/local/lib/python3.4/site-packages/pyfinder-0.2-py3.4.egg-info
```

Per poterla installare, l'utente dovrà quindi avere lo stesso sistema operativo utilizzato per creare la distribuzione e anche la medesima struttura di directory. L'installazione, in questo caso, consiste nel copiare la distribuzione nella root e scompattarla:

```
$ sudo cp dist/pyfinder-0.2.linux-i686.tar.gz /
$ cd /
$ sudo tar zxvf pyfinder-0.2.linux-i686.tar.gz
./
./usr/
./usr/local/
./usr/local/bin/
./usr/local/bin/pyfinder
./usr/local/lib/
./usr/local/lib/python3.4/
./usr/local/lib/python3.4/site-packages/
./usr/local/lib/python3.4/site-packages/pyfinder-0.2-py3.4.egg-info
./usr/local/lib/python3.4/site-packages/pyfinder.py
./usr/local/lib/python3.4/site-packages/__pycache__/
./usr/local/lib/python3.4/site-packages/__pycache__/pyfinder.cpython-34.pyc
```

Anche questa volta possiamo scegliere il formato con l'opzione `--formats`:

```
python setup.py bdist --format=zip
$ ls dist/*.zip
dist/pyfinder-0.2.linux-i686.zip dist/pyfinder-0.2.zip
```

Oltre ai formati che abbiamo visto nel caso della creazione della source distribution, in questo caso sono disponibili anche altri formati, come, ad esempio, `rpm`, `wininst` (self-extracting ZIP file per Windows) e `msi` (Microsoft Installer). Ad esempio, se in un sistema operativo Windows a 32 bit diamo il

comando:

```
> python setup.py bdist --formats=msi
```

viene creato l'installer `pyfinder-0.2.win32.msi`. Quando questo viene eseguito, si apre un wizard di installazione, come illustrato in [Figura 4.2](#).

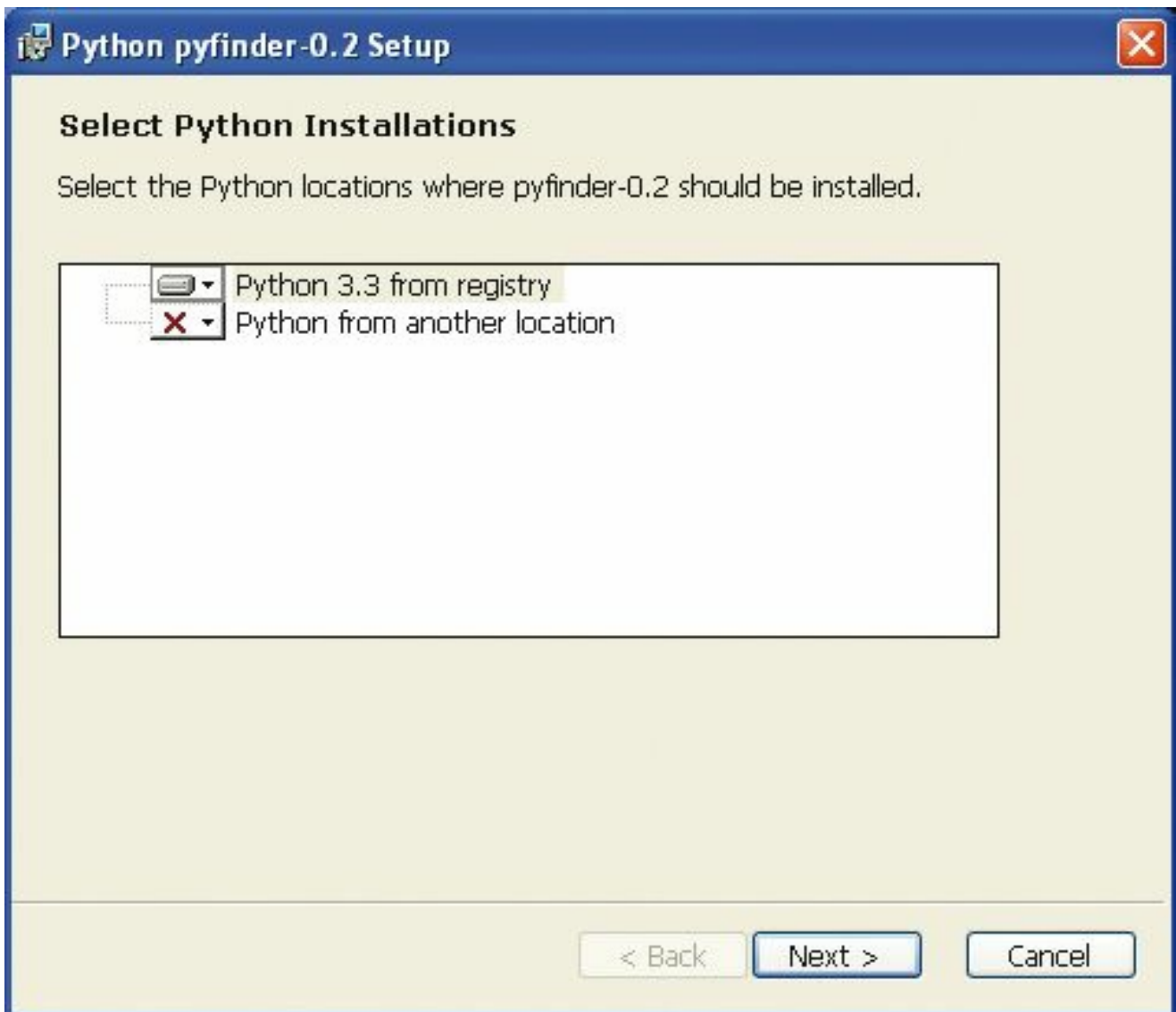


Figura 4.2 - Wizard di installazione dell'applicazione *pyfinder* su Windows.

NOTA

Abbiamo visto come creare delle distribuzioni che possono essere utilizzate da utenti che hanno una installazione di Python funzionante nel loro sistema. Spesso, però, si ha l'esigenza di distribuire delle applicazioni per utenti che non hanno una installazione di Python. Questo tipo di applicazioni, che non necessitano di dipendenze, sono dette *stand-alone*. Esistono vari tool che consentono di creare applicazioni di questo tipo, come ad esempio

Pubblicare la distribuzione su PyPI

Come abbiamo detto più volte, pubblicare la distribuzione significa renderla disponibile agli utenti. Il modo più semplice per farlo, che tra l'altro facilita anche il processo di installazione dell'applicazione da parte degli utenti, consiste nel registrare e caricare l'applicazione su PyPI, utilizzando i comandi `register` e `upload` del modulo `distutils`.

Registrare l'applicazione su PyPI

Il comando `register` di `distutils` invia i metadati dell'applicazione a PyPI. Registriamo, quindi, PyFinder:

```
$ python setup.py register
running register
running check
We need to know who you are, so please choose either:
1. use your existing login,
2. register as a new user,
3. have the server generate a new password for you (and email it to you), or
4. quit
Your selection [default 1]:
2
```

Come possiamo vedere, ci viene mostrato un prompt con quattro possibili scelte. Se non ci siamo ancora registrati su PyPI, usiamo l'opzione 2, altrimenti indichiamo la 1 e inseriamo nome utente e password. Ci verrà chiesto di indicare anche un indirizzo E-mail, che verrà utilizzato per inviarci un messaggio tramite il quale potremo completare la registrazione.

Una volta registrati, possiamo nuovamente eseguire il comando di registrazione. Questa volta scegliamo l'opzione 1 e inseriamo il nome utente e la password:

```
$ python setup.py register
running register
running check
We need to know who you are, so please choose either:
1. use your existing login,
2. register as a new user,
3. have the server generate a new password for you (and email it to you), or
4. quit
Your selection [default 1]:
1
Username: marco.buttu
```

```
Password:
Registering pyfinder to http://pypi.python.org/pypi
Server response (200): OK
I can store your PyPI login so future submissions will be faster.
(the login will be stored in /home/marco/.pypirc)
Save your login (y/N)?y
```

Come possiamo vedere, ci viene chiesto se vogliamo salvare nel file *.pypirc* le credenziali di accesso a PyPI, in modo da evitare di inserire nome-utente e password durante gli accessi futuri.

Una volta completata la registrazione, viene creata la pagina web <https://pypi.python.org/pypi/pyfinder/>, nella quale troviamo i metadati di PyFinder ma non la distribuzione, visto che dobbiamo ancora caricarla con il comando `upload`.

NOTA

PyPI crea l'indirizzo della pagina web di un'applicazione aggiungendo il nome di questa in coda all'URL <https://pypi.python.org/pypi/>.

Caricare l'applicazione su PyPI

Per caricare l'applicazione su PyPI utilizziamo il comando `upload`. Questo esegue due operazioni: crea la distribuzione e la invia a PyPI. È quindi necessario indicare il tipo di distribuzione che vogliamo pubblicare. Nel nostro caso siamo interessati a una source distribution, per cui diamo il comando:

```
$ python setup.py sdist upload
```

Come possiamo vedere in [Figura 4.3](#), la distribuzione è ora disponibile per il download.

A questo punto gli utenti potranno installare PyFinder in modo semplice, tramite *pip*; facciamo una prova con il nostro ambiente virtuale:

```
(myenv) $ source ~/myenv/bin/activate
(myenv) $ pip install pyfinder
Downloading/unpacking pyfinder
Downloading pyfinder-0.2.tar.gz
Running setup.py egg_info for package pyfinder

Installing collected packages: pyfinder
Running setup.py install for pyfinder
changing mode of build/scripts-3.4/pyfinder from 644 to 755
```

changing mode of /home/marco/myvenv/bin/pyfinder to 755
Successfully installed pyfinder
Cleaning up...

E usare lo script `pyfinder` da linea di comando.

Con questo esercizio abbiamo concluso la prima parte del libro. Nei prossimi due capitoli parleremo in dettaglio di classi e di programmazione orientata agli oggetti.



Figura 4.3 - La pagina web di PyFinder sul Python Package Index.

Classi e programmazione orientata agli oggetti

Questo capitolo è dedicato alla **programmazione a oggetti** con Python. Parleremo di **classi e istanze**, **attributi**, **ereditarietà**, **overloading**, **decoratori** e **gestione delle eccezioni**. Nell'esercizio conclusivo vedremo un esempio di utilizzo del modulo *cmd* della libreria standard.

Classi e istanze

Se un oggetto creato con l'istruzione `def` è detto *funzione*, uno creato tramite l'istruzione `class` è detto *classe*. Le classi sono gli elementi che stanno alla base della *programmazione orientata agli oggetti* (OOP, Object Oriented Programming).

Una classe viene definita facendo seguire la parola chiave `class` dall'etichetta tramite la quale si vuole fare riferimento a essa:

```
>>> class Foo:
...     pass
...
>>> Foo
<class '__main__.Foo'>
```

Come per le funzioni, le classi vengono create a *runtime*, quando l'istruzione `class` viene eseguita.

A differenza delle funzioni, il cui codice viene eseguito solo al momento della chiamata:

```
>>> def foo():
...     print('python')
...     import sys
...     print(sys.platform)
...
>>> foo()
python
linux
```

il codice contenuto all'interno della classe viene eseguito immediatamente al momento della sua definizione:

```
>>> class Foo:
...     print('python')
...     import sys
...     print(sys.platform)
...
python
linux
```

Quindi, come abbiamo detto nel Capitolo 4, ogni etichetta libera nel corpo della classe viene risolta immediatamente al momento della definizione:

```
>>> class Foo:
...     print(a)
...
Traceback (most recent call last):
...
NameError: name 'a' is not defined
>>> a = 99
>>> class Foo:
...     print(a)
...
99
```

Le classi hanno una serie di attributi detti *attributi speciali* o *attributi magici*, che iniziano e finiscono con due underscore. Ad esempio, `__name__` è un attributo speciale, al quale viene assegnato il nome dell'etichetta utilizzata nella definizione della classe:

```
>>> Foo.__name__
'Foo'
```

L'attributo `__module__` è anch'esso speciale e fa riferimento al nome del modulo nel quale è definita la classe:

```
>>> Foo.__module__ # Foo è definita nella shell interattiva, ovvero nel modulo __main__
'__main__'
```

NOTA

Le classi convenzionalmente vengono definite utilizzando delle etichette che iniziano con una lettera maiuscola.

Le classi vengono utilizzate per definire dei *tipi di dato*. I tipi di dato built-in che abbiamo visto nel Capitolo 1 (`int`, `float`, `complex`, `str`, `list`, `tuple`, `dict` e `set`) sono delle classi. Ad esempio, il tipo di dato `list` è una classe tramite la quale possiamo creare degli oggetti di tipo `list`, i quali si comportano tutti allo stesso modo, ovvero hanno gli stessi *attributi*. Gli oggetti creati tramite la classe sono detti *istanze della classe*. Come sappiamo, per i tipi di dato built-in il linguaggio definisce dei *letterali* tramite i quali è possibile creare le istanze. Ad esempio, nel caso delle liste si utilizzano le parentesi quadre:

```
>>> list1 = [1, 2, 'python']
>>> list2 = [('a', 'b'), 44.5]
```

Poiché una classe definisce un tipo di dato, utilizzeremo le parole *tipo* e *classe* in modo intercambiabile, per cui diremo che un'istanza di una classe `list` è un'istanza del *tipo* `list`. Allo stesso modo, di un *oggetto di tipo* `list` diremo che è un *oggetto di classe* `list`. Possiamo conoscere il tipo di un oggetto sia tramite il suo attributo `__class__` sia tramite la classe built-in `type`:

```
>>> list1.__class__, list2.__class__
(<class 'list'>, <class 'list'>)
>>> type(list1), type(list2)
(<class 'list'>, <class 'list'>)
```

Inoltre possiamo utilizzare la funzione built-in `isinstance()` per sapere se un oggetto è una istanza di una data classe:

```
>>> mylist = [1, 2, 3]
>>> isinstance(mylist, list), isinstance(mylist, tuple)
(True, False)
```

Se le istanze dei tipi built-in possono essere create tramite i loro letterali, questo non è possibile per le altre classi, poiché per esse non esistono dei letterali. La sintassi generale che si utilizza per creare le istanze è quella di *chiamare* la classe, allo stesso modo di come si chiama una funzione:

```
>>> class Foo:
...     pass
...
>>> callable(Foo) # La classe è un oggetto chiamabile
True
>>> f = Foo() # Quando una classe viene chiamata, viene creata una sua istanza
>>> g = Foo()
>>> type(f), type(g)
(<class '__main__.Foo'>, <class '__main__.Foo'>)
>>> f.__class__
<class '__main__.Foo'>
```

NOTA

È indicato `<class '__main__.Foo'>` piuttosto che `<class 'Foo'>` perché la classe è definita nel modulo `__main__` (stiamo lavorando in modo interattivo).

Questa sintassi è generale e quindi vale anche per i tipi built-in:

```
>>> list1 = list()
>>> list2 = list((1, 2, 3, 'ciao'))
>>> list1
[]
>>> list2
[1, 2, 3, 'ciao']
```

Poiché abbiamo detto che una classe è un tipo di dato e che useremo le parole *classe* e *tipo* in modo intercambiabile, è facile prevedere di che tipo è una classe:

```
>>> type(Foo), Foo.__class__
(<class 'type'>, <class 'type'>)
>>> type(list), list.__class__
(<class 'type'>, <class 'type'>)
```

Quindi tutte le classi sono oggetti (istanze) di tipo `type`, mentre le istanze di una generica classe `Foo` sono oggetti di tipo (o classe) `Foo`:

```
>>> class Foo:
...     pass
...
>>> f = Foo()
>>> type(f)
<class '__main__.Foo'>
>>> type(Foo)
<class 'type'>
```

Chiariremo meglio questi concetti nel [Capitolo 6](#).

Il parametro `self`

Come abbiamo già detto nel [Capitolo 1](#), gli attributi possono essere classificati a seconda che siano chiamabili oppure no. Se sono chiamabili, allora vengono detti *metodi*.

La seguente classe, ad esempio, ha un attributo `a` che non è chiamabile e un metodo `Foo.foo_method()` che prende un oggetto e ne stampa l'identificativo:

```
>>> class Foo:
...     a = 33
...     def foo_method(self):
...         print(id(self))
...
>>> Foo.a
33
>>> mylist = [1, 2, 3]
```



```
>>> Foo.foo_method(mylist)
3072341196
>>> id(mylist)
3072341196
```

I metodi tipicamente eseguono delle operazioni sulle istanze della classe, per cui solitamente ricevono l'istanza come argomento per poi processarla:

```
>>> f = Foo()
>>> Foo.foo_method(f)
3072442028
>>> id(f)
3072442028
```

Passare l'istanza al metodo è un'operazione che sta alla base della programmazione orientata agli oggetti, e per questo motivo è stata definita una sintassi alternativa che consente di passare l'istanza al metodo in modo automatico. In base a questa convenzione, quando un metodo è chiamato direttamente tramite l'istanza, questa viene passata al metodo come *primo* argomento. Quindi la chiamata `Foo.foo_method(f)` ha lo stesso effetto della seguente:

```
>>> f.foo_method()
3072442028
```

NOTA

Come vedremo più avanti, questo comportamento non è sempre vero, perché esiste una tipologia di metodi, detti metodi di classe, a cui viene sempre passata la classe come primo argomento, anche quando essi vengono qualificati tramite l'istanza.

In questo caso, come abbiamo potuto vedere, l'istanza `f` è stata passata a `Foo.foo_method()`. Ecco, infatti, cosa accade se si tenta di chiamare tramite l'istanza un metodo che non accetta argomenti:

```
>>> class Foo:
...     def foo_method():
...         print('python')
...
>>> Foo.foo_method()
python
>>> f = Foo()
>>> f.foo_method() # Equivale a `Foo.foo_method(f)`
```

Traceback (most recent call last):

...

TypeError: foo_method() takes 0 positional arguments but 1 was given

Il parametro al quale viene assegnata l'istanza viene convenzionalmente chiamato `self`, come indicato nella sezione *Function and method arguments* della PEP-0008.

NOTA

Per scoprire le motivazioni che hanno portato alla decisione di utilizzare il parametro `self` in modo esplicito, possiamo visitare l'URL <http://docs.python.org/3/faq/design.html> e leggere la sezione *Why must 'self' be used explicitly in method definitions and calls?*

L'etichetta speciale `__class__`, usata all'interno di un metodo, fa riferimento alla classe:

```
>>> class Foo:
...     def foo():
...         print(__class__)
...
>>> Foo.foo()
<class '__main__.Foo'>
```

Attributi di classe e di istanza

Per quanto riguarda il loro ambito di visibilità (lo scope), gli attributi possono essere di classe oppure di istanza. Vediamo di chiarire questo concetto. Gli assegnamenti fatti al di fuori dei metodi, ovvero al *top-level* della classe (primo livello di indentazione), creano degli *attributi di classe*, cioè attributi che sono condivisi da tutte le istanze:

```
>>> class Foo:
...     a = 33
...     def foo_method(self, obj):
...         Foo.a = obj
...
>>> Foo.a
33
>>> f = Foo()
>>> g = Foo()
>>> f.a, g.a
```

(33, 33)

Ciò significa che se una classe o una istanza modifica un attributo di classe, tale modifica viene vista sia dalla classe sia da tutte le altre sue istanze:

```
>>> Foo.a = 'python'
>>> f.a, g.a
('python', 'python')
>>> f.foo_method([1, 2, 3])
>>> g.a
[1, 2, 3]
>>> Foo.a
[1, 2, 3]
```

Anche i metodi sono degli attributi di classe, in quanto definiti al top-level della classe, per cui sono condivisi da tutte le istanze:

```
>>> def foo(self):
...     print(id(self))
...
>>> Foo.foo_method = foo
>>> f.foo_method()
140128939814736
>>> g.foo_method()
140128939814800
```

Gli assegnamenti fatti a etichette qualificate tramite l'istanza creano degli *attributi di istanza*, ovvero particolari di quella istanza e non condivisi né con altre istanze né, tanto meno, con la classe:

```
>>> class Foo:
...     pass
...
>>> f = Foo()
>>> g = Foo()
>>> f.a = 33
>>> hasattr(Foo, 'a')
False
>>> hasattr(g, 'a')
False
>>> hasattr(f, 'a')
True
>>> f.a
33
```

Gli assegnamenti fatti a etichette qualificate tramite la classe creano attributi di classe, condivisi quindi da tutte le istanze:

```

class Foo:
... a = 33 # Attributo di classe
...
>>> f = Foo()
>>> g = Foo()
>>> f.a, g.a
(33, 33)
>>> Foo.b = 100 # Crea un attributo di classe
>>> f.b, g.b
(100, 100)

```

Se all'interno di un metodo vogliamo creare un attributo di istanza, dobbiamo qualificare l'attributo tramite il parametro che fa riferimento all'istanza:

```

>>> class Foo:
... a = 33 # Attributo di classe
... def foo_method(self, x):
... self.b = x
...
>>> f = Foo()
>>> g = Foo()
>>> hasattr(Foo, 'b'), hasattr(f, 'b'), hasattr(g, 'b')
(False, False, False)
>>> f.foo_method('ffffff')
>>> hasattr(Foo, 'b'), hasattr(f, 'b'), hasattr(g, 'b')
(False, True, False)
>>> f.b
'ffffff'
>>> g.foo_method('gggggg')
>>> hasattr(Foo, 'b'), hasattr(f, 'b'), hasattr(g, 'b')
(False, True, True)
>>> g.b, f.b
('gggggg', 'ffffff')
>>> Foo.b = 'Foooloooo' # Creo un attributo di classe
>>> hasattr(Foo, 'b'), hasattr(f, 'b'), hasattr(g, 'b')
(True, True, True)
>>> Foo.b, f.b, g.b
('Foooloooo', 'ffffff', 'gggggg')

```

Quindi la chiamata `f.foo_method('ffffff')` esegue l'istruzione `self.b = 'ffffff'`, la quale è esattamente equivalente a `f.b = 'ffffff'`, visto che `self` e `f` sono due etichette che fanno riferimento al medesimo oggetto.

Allo stesso modo, se da un metodo vogliamo fare riferimento a un attributo di istanza, dobbiamo qualificare l'attributo con il riferimento all'istanza:

```

>>> class Foo:
... def methodA(self, value):
... self.value = value

```

```
... def methodB(self):
...     print(self.value)
...
>>> f = Foo()
>>> f.methodA('python')
>>> f.methodB()
Python
```

Concludiamo questa sezione con una osservazione. Consideriamo il seguente esempio:

```
>>> class Foo:
...     def __init__(self, data=[]):
...         self.data = data
...
>>> a = Foo()
>>> b = Foo()
>>> a.data.append(1)
>>> a.data
[1]
>>> b.data
[1]
```

Sembrerebbe che `a.data` e `b.data` siano attributi di classe, ma così non è:

```
>>> Foo.data
Traceback (most recent call last):
...
AttributeError: type object 'Foo' has no attribute 'data'
```

Infatti, come abbiamo visto nel [Capitolo 3](#), questo comportamento è legato all'utilizzo in `Foo.__init__()` di un argomento di default mutabile. Come sappiamo, non viene creato un nuovo oggetto di default a ogni chiamata, ma viene utilizzato sempre lo stesso, per cui `a.data` e `b.data` sono due attributi di istanza, ma con la particolarità di fare riferimento al medesimo oggetto:

```
>>> a.data is b.data
True
```

esattamente come nel caso seguente:

```
>>> class Foo:
...     def __init__(self, data):
...         self.data = data
...
>>> mylist = []
```

```
>>> a = Foo(mylist)
>>> b = Foo(mylist)
>>> a.data.append(1)
>>> b.data
[1]
```

Questo comportamento, talvolta, è causa di errori logici, poiché, essendo in generale controintuitivo, se non lo si conosce porta a fare delle assunzioni errate. Una soluzione potrebbe essere quella di effettuare una copia dell'oggetto:

```
>>> class Foo:
...     def __init__(self, data):
...         self.data = data.copy()
...
>>> mylist = []
>>> a = Foo(mylist)
>>> b = Foo(mylist)
>>> a.data.append(1)
>>> b.data
[]
```

Si osservi che, in questo caso, viene sempre fatta una copia, anche quando non sarebbe necessario.

Scope e namespace

Gli attributi di classe e di istanza sono locali:

```
>>> class Foo:
...     a = 33
...     def foo(self):
...         self.b = 100
...
>>> 'a' in globals(), 'foo' in globals(), 'b' in globals()
(False, False, False)
>>> 'a' in vars(Foo), 'foo' in vars(Foo), 'b' in vars(Foo)
(True, True, False)
>>> f = Foo()
>>> f.foo()
>>> 'a' in vars(f), 'foo' in vars(f), 'b' in vars(f)
(False, False, True)
```

Lo scope locale a una istanza è *enclosing* rispetto allo scope locale della sua classe, per cui l'istanza continua a vedere gli attributi di classe anche se questa non esiste più:

```
>>> del Foo
>>> f.a
33
>>> f.foo
<bound method Foo.foo of <__main__.Foo object at 0xb738724c>>
```

Gli attributi qualificati tramite l'istanza nascondono quelli omonimi qualificati tramite la classe:

```
>>> class Foo:
...     a = 33
...
>>> f = Foo()
>>> g = Foo()
>>> f.a
33
>>> f.a = 'python'
>>> f.a, g.a
('python', 33)
```

Lo scope di classe non è enclosing rispetto allo scope locale dei metodi:

```
>>> class Foo:
...     c = 33
...     def foo():
...         print(c)
...
>>> Foo.foo()
Traceback (most recent call last):
...
NameError: global name 'c' is not defined
```

I metodi, infatti, sono delle funzioni e, come tali, risolvono le etichette libere prima di tutto in uno scope enclosing (lo scope di una funzione che li racchiude):

```
>>> def foo():
...     a = 'python3.4'
...     class Foo:
...         def moo():
...             print(a) # Viene risolta nello scope locale di `foo()`
...             return Foo
...
>>> F = foo()
>>> F.moo()
python3.4
```

altrimenti nello scope globale:

```
>>> class Foo:
...     def foo():
...         print(d)
...
>>> Foo.foo()
Traceback (most recent call last):
...
NameError: global name 'd' is not defined
>>> d = 66
>>> Foo.foo()
66
```

Quindi, se si vuole utilizzare un attributo di classe all'interno di un metodo, è necessario qualificarlo con l'etichetta di classe:

```
>>> class Foo:
...     c = 33
...     def foo():
...         print(Foo.c)
...
>>> Foo.foo()
33
```

Quando all'interno di un metodo viene fatto un assegnamento a una etichetta non qualificata, allora viene creata una etichetta locale al metodo:

```
>>> class Foo:
...     def foo_method():
...         a = 99
...         print(locals())
...
...
>>> Foo.foo_method()
{'a': 99}
```

Le etichette locali, come sappiamo, vengono cancellate quando il metodo termina la sua esecuzione.

Riassumendo, gli assegnamenti fatti al top-level della classe creano *attributi di classe*, quelli fatti a etichette qualificate tramite il riferimento all'istanza creano *attributi di istanza*, quelli fatti all'interno dei metodi a etichette non qualificate creano *etichette locali* al metodo:


```

class Foo:
... a = 1 # Attributo di classe
... def foo(self):
... self.b = 2 # Attributo di istanza
... c = 99 # Etichetta locale
...
>>> Foo.a
1
>>> f = Foo()
>>> f.foo()
>>> f.b
2
>>> f.c
Traceback (most recent call last):
...
AttributeError: 'Foo' object has no attribute 'c'

```

Il namespace delle classi, così come quello dei moduli, è implementato tramite un dizionario, assegnato all'attributo `__dict__` della classe:

```

>>> class Foo:
... a = 44
...
>>> 'a' in Foo.__dict__
True
>>> 'b' in Foo.__dict__
False
>>> Foo.b = 33
>>> 'b' in Foo.__dict__
True

```

Anche il namespace delle istanze è implementato tramite un dizionario:

```

>>> f = Foo()
>>> f.__dict__
{}
>>> f.c = 100
>>> f.__dict__
{'c': 100}
>>> f.c
100
>>> f.__dict__['c']
100
>>> del f.__dict__['c']
>>> f.c
Traceback (most recent call last):
...
AttributeError: 'Foo' object has no attribute 'c'
>>> f.__dict__['d'] = 200
>>> f.d
200

```

Le due notazioni, però, non sono equivalenti:

```
>>> f.a
44
>>> f.__dict__['a']
Traceback (most recent call last):
...
KeyError: 'a'
```

Infatti gli attributi di istanza vengono cercati, prima di tutto, nel namespace dell'istanza, per cui `f.a` viene prima cercato in `f.__dict__`:

```
>>> 'a' in f.__dict__
False
>>> f.__dict__
{'d': 200}
```

Quando un attributo di istanza non è presente nel suo dizionario degli attributi, allora, se risulta che l'istanza ha comunque l'attributo, questo viene cercato nel dizionario degli attributi della classe. Nel nostro caso, quindi, l'attributo `a` non è presente in `f.__dict__`, però l'istanza ha l'attributo:

```
>>> hasattr(f, 'a')
True
```

quindi il look-up di `f.a` continua cercando l'attributo `a` in `Foo.__dict__`:

```
>>> 'a' in Foo.__dict__
True
>>> Foo.__dict__['a']
44
```

NOTA

Per versioni di Python precedenti alla 3.3, nel caso in cui si vogliano creare delle classi da utilizzare principalmente come strutture dati, che devono istanziare un elevato numero di istanze, si può scegliere una diversa implementazione del namespace, realizzata mediante un vettore di dimensioni fisse, piuttosto che tramite dizionario. Per fare ciò si utilizza l'attributo speciale `__slots__`, nel modo seguente:

```
>>> class Point:
...     __slots__ = ('x', 'y', 'z')
...     def __init__(self, x, y, z):
```

```
... self.x = x
... self.y = y
... self.z = z
```

La PEP-0412 risolve il problema dell'ottimizzazione e rende superfluo l'utilizzo di `__slots__`. In ogni caso, per versioni di Python precedenti alla 3.3, anche quando può avere senso utilizzarlo, bisogna pensarci su più di una volta, perché in generale chi usa la classe si aspetta che il namespace sia implementato tramite dizionario.

Approfondiremo il discorso sulla ricerca degli attributi (*attribute lookup*) sia quando parleremo del meccanismo dell'ereditarietà, più avanti in questo capitolo, sia nel [Capitolo 6](#), nella sezione intitolata *Accesso agli attributi*.

Differenze di implementazione tra il namespace di classe e di istanza

Come abbiamo detto nella sezione precedente, sia il namespace di classe sia quello di istanza sono implementati tramite un dizionario. Il dizionario degli attributi di istanza è un dizionario ordinario (un oggetto di tipo `dict`):

```
>>> class Foo:
...     a = 33
...
>>> f = Foo()
>>> type(f.__dict__)
<class 'dict'>
```

Il dizionario degli attributi di classe non è, invece, un dizionario ordinario, bensì un oggetto di tipo `types.MappingProxyType`:

```
>>> import types
>>> types.MappingProxyType is type(Foo.__dict__)
True
```

A differenza di un dizionario ordinario, un oggetto di tipo `MappingProxyType` non supporta l'assegnamento dei suoi elementi:

```
>>> f.__dict__['b'] = 44
>>> Foo.__dict__['c'] = 55
Traceback (most recent call last):
...
TypeError: 'mappingproxy' object does not support item assignment
```

Un dizionario di questo tipo non può neppure essere aggiornato:

```
>>> f.__dict__.update({'d': 66})
>>> Foo.__dict__.update({'e': 77})
Traceback (most recent call last):
...
AttributeError: 'mappingproxy' object has no attribute 'update'
```

Se vogliamo modificare il dizionario degli attributi di classe dobbiamo usare la funzione built-in `setattr()`:

```
>>> setattr(Foo, 'foo', 88)
>>> 'foo' in Foo.__dict__
True
>>> Foo.foo
88
```

Come possiamo vedere, `setattr()` prende un oggetto come primo argomento, il nome di un attributo come secondo argomento e l'oggetto da assegnare all'attributo come terzo argomento. Quindi, quando abbiamo il nome di un attributo (una stringa), se vogliamo ottenere il valore usiamo `getattr()`, mentre, se vogliamo assegnare un valore, usiamo `setattr()`.

NOTA

A differenza di Python 3, in Python 2 il dizionario degli attributi di classe è anch'esso un normale dizionario ordinario:

```
>>> class Foo: # Python 2.7
...     pass
...
>>> type(Foo.__dict__)
<type 'dict'>
```

Attributi nascosti

In Python non esistono *specificatori di accesso* agli attributi, come il *public*, *private* e *protected* utilizzati in altri linguaggi, e gli attributi sono sempre visibili dall'esterno (*public*). È possibile, però, *nascondere* un attributo di classe. Per fare ciò lo si fa iniziare, ma non terminare, con due underscore:

```
>>> class Foo:
...     __a = 99
...
```

```
>>> hasattr(Foo, '__a')
False
>>> Foo.__a
Traceback (most recent call last):
...
AttributeError: type object 'Foo' has no attribute '__a'
```

L'attributo, in realtà, è visibile, ma ha un altro nome. Infatti, quando si assegna un oggetto a un attributo di classe che inizia ma non finisce con due underscore, Python lo modifica automaticamente facendolo precedere da un underscore seguito dal nome della classe:

```
>>> hasattr(Foo, '_Foo__a')
True
>>> Foo._Foo__a
99
```

Vengono nascosti anche gli attributi di istanza:

```
>>> class Foo:
...     def __init__(self, a):
...         self.__a = a
...
>>> f = Foo(100)
>>> f.__a
Traceback (most recent call last):
...
AttributeError: 'Foo' object has no attribute '__a'
>>> f._Foo__a
100
```

Infine, osserviamo come questa trasformazione possa riservare delle sorprese:

```
>>> class Foo:
...     _Foo__a = 33
...     __a = 'python'
...
>>> Foo._Foo__a
'python'
```

NOTA

Se siamo curiosi di conoscere il motivo per cui in Python non esistono specificatori di accesso, come il *public* e *private* di altri linguaggi, possiamo cercare in Internet la frase *We're all consenting adults here* e sicuramente troveremo risposta alle nostre domande.

L'inizializzatore

Un attributo di istanza esiste solo a partire dal momento in cui viene assegnato. La seguente definizione crea una classe `Foo` e i suoi attributi di classe a `e mymethod`:

```
>>> class Foo:
...     a = 33
...     def mymethod(self):
...         self.b = 44
...
>>> f = Foo()
>>> f.a
33
```

I metodi sono delle funzioni, per cui, come abbiamo visto nei precedenti capitoli, le istruzioni al loro interno vengono eseguite solo a *runtime*, quando il metodo viene chiamato. Per questo motivo l'istanza `f` non ha ancora l'attributo `b`, poiché questo viene creato solo dopo la chiamata `f.mymethod()`:

```
>>> hasattr(f, 'b')
False
>>> f.b
Traceback (most recent call last):
...
AttributeError: 'Foo' object has no attribute 'b'
>>> f.mymethod()
>>> f.b
44
```

Spesso si vuole che alcuni attributi di istanza siano disponibili da subito, poiché devono essere utilizzati da vari metodi e quindi devono essere già definiti quando questi metodi vengono chiamati. Per fare ciò si definisce un metodo speciale, indicato con `__init__()` e detto *inizializzatore*, il quale viene chiamato automaticamente da Python dopo la creazione dell'istanza, al fine di inicializzarla:

```
>>> class Foo:
...     def __init__(self):
...         self.a = 33
...
>>> f = Foo() # Viene creata l'istanza e poi chiamato automaticamente `f.__init__()`
>>> f.a # Infatti, come possiamo vedere, l'attributo `a` esiste già
33
```

Quando un inizializzatore accetta degli argomenti, questi vanno passati al momento della creazione dell'istanza, elencandoli tra le parentesi che seguono l'etichetta della classe:

```
>>> from math import pi
>>> class Circle:
...     def __init__(self, r=1):
...         self.radius = 0 if r < 0 else r
...     def area(self):
...         return pi * self.radius**2
...
>>> c = Circle(3) # Crea un oggetto di tipo `Circle` avente raggio pari a 3
>>> c.area()
28.274333882308138
>>> c = Circle() # Crea un cerchio di raggio 1 (valore di default)
>>> c.area()
3.141592653589793
```

Nel Capitolo 6 parleremo in dettaglio sia della creazione sia dell'inizializzazione delle istanze.

Un primo sguardo all'overloading

Nel corso del libro abbiamo utilizzato vari operatori per eseguire delle operazioni sui tipi built-in. Abbiamo visto, ad esempio, che l'operatore `+` consente di sommare dei numeri, oppure di concatenare delle sequenze:

```
>>> 22 + 33
55
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

mentre l'operatore `*` esegue il prodotto per i numeri e la ripetizione per le sequenze:

```
>>> 22 * 3
66
>>> 'python' * 3
'pythonpythonpython'
```

Questo è possibile perché Python sa come eseguire queste operazioni con i tipi built-in. Quando si definisce una classe, se si vogliono effettuare delle operazioni con le sue istanze è necessario indicare come farlo. Consideriamo, ad esempio, la classe `Circle` definita nella sezione precedente. Se proviamo a sommare due sue istanze otteniamo un errore, perché Python non sa come fare questa somma:

```
>>> c1 = Circle(10)
>>> c2 = Circle(20)
>>> c1 + c2
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Circle' and 'Circle'
```

Per poter effettuare questa operazione, dobbiamo prima di tutto decidere che significato vogliamo dare alla somma di due oggetti di tipo `Circle`. Supponiamo, ad esempio, di voler fare in modo che la somma crei un nuovo oggetto di tipo `Circle` con raggio pari alla somma dei raggi. L'operazione di somma va definita in un metodo speciale, chiamato `__add__()`:

```
>>> import math
>>> class Circle:
```



```

... def __init__(self, r=1):
...     self.radius = 0 if r < 0 else r
... def __add__(self, other):
...     return Circle(self.radius + other.radius)
... def area(self):
...     return math.pi * self.radius**2
...
>>> c1 = Circle(10)
>>> c2 = Circle(20)
>>> c = c1 + c2
>>> c.radius
30
>>> c.area()
2827.4333882308138

```

Vediamo di chiarire il precedente codice. Quando Python incontra l'operazione `c1 + c2`, esegue automaticamente la chiamata `c1.__add__(c2)`, ovvero `Circle.__add__(c1, c2)`. Quindi il simbolo `+` è solamente una rappresentazione grafica alla quale Python fa corrispondere la chiamata al metodo `Circle.__add__()`, passando a questo l'istanza alla sinistra del `+` come primo argomento e quella a destra come secondo. Quindi nel nostro caso `c1` è stato assegnato all'etichetta `self`, mentre `c2` a `other`. Il metodo `Circle.__add__()` fa poi la chiamata a `Circle(self.radius + other.radius)`, la quale crea un nuovo oggetto di tipo `Circle` di raggio pari a `self.radius + other.radius`, il quale viene restituito con `return`.

NOTA

Nella sezione *Overloading degli operatori aritmetici ed ereditarietà* del Capitolo 6 vedremo come, nel caso in cui ereditiamo da `Circle` e non ridefiniamo il metodo `Circle.__add__()`, allora `Circle.__add__()` deve restituire `self.__class__(self.radius + other.radius)` piuttosto che `Circle(self.radius + other.radius)`.

Python a ogni rappresentazione grafica di un operatore associa un metodo da chiamare. Ad esempio, `c1 * c2` dà luogo alla chiamata `c1.__mul__(c2)`, `c1 - c2` comporta la chiamata `c1.__sub__(c2)`, e così via:

```

>>> import math
>>> class Circle:
...     def __init__(self, r=1):
...         self.radius = 0 if r < 0 else r
...     def __add__(self, other):

```

```

... return Circle(self.radius + other.radius)
... def __mul__(self, other):
...     return Circle(self.radius * other.radius)
... def __sub__(self, other):
...     return Circle(self.radius - other.radius)
... def area(self):
...     return math.pi * self.radius**2
...
>>> c1 = Circle(10)
>>> c2 = Circle(20)
>>> c = c1 + c2
>>> c.radius
30
>>> c = c1 * c2
>>> c.radius
200
>>> c = c2 - c1
>>> c.radius
10
>>> c = c1 - c2
>>> c.radius
0

```

Questo meccanismo, detto *overloading*, ci consente di utilizzare i classici operatori aritmetici per effettuare delle operazioni tra istanze, rendendo il codice più snello e facile da leggere rispetto al codice che avremmo dovuto utilizzare facendo un uso diretto dei metodi:

```

>>> c = (c1 + c2) * c1 # Utilizzo degli operatori aritmetici
>>> c.radius
300
>>> c = (c1.__add__(c2)).__mul__(c1) # Utilizzo diretto dei metodi
>>> c.radius
300

```

Approfondiremo l'argomento nella sezione *Attributi speciali* del Capitolo 6.

La composizione

Un importante concetto della OOP è la cosiddetta *composizione*. Questo termine viene utilizzato per indicare che un oggetto è composto da un altro oggetto, ovvero che un attributo di un oggetto di tipo *A* fa riferimento a un oggetto di tipo *B*. Questo in Python è un meccanismo naturale, poiché, come ormai sappiamo, ogni etichetta (e quindi ogni attributo) fa riferimento a un oggetto.

Tipicamente la relazione di composizione viene utilizzata quando possiamo dire che un oggetto *ha un* altro oggetto, e infatti nella lingua inglese è anche chiamata relazione *has a*. Vediamo di chiarire questo concetto. Riprendiamo, a tale proposito, la classe `Circle`. Come sappiamo, un cerchio *ha un* centro, individuato dalle sue coordinate *x* e *y* nel piano cartesiano: questo è proprio un esempio di relazione *has a* (“*a circle has a center*”). Il centro può essere modellato con una generica classe `Point`:

```
>>> import math
>>> class Point:
...     def __init__(self, x=0, y=0):
...         self.x = x
...         self.y = y
...     def distance(self):
...         """Restituisci la distanza dall'origine"""
...         return math.sqrt(self.x**2 + self.y**2)
...     def __str__(self):
...         """Restituisci una rappresentazione del punto sotto forma di stringa."""
...         return str((self.x, self.y))
```

Il metodo `__str__()` ha un significato speciale. Viene automaticamente chiamato da Python nei casi in cui si vuole ottenere una rappresentazione di un oggetto `obj` sotto forma di stringa, ovvero quando si effettua la chiamata `str(obj)`.

NOTA

Abbiamo visto, nella sezione *La funzione built-in print()* del Capitolo 1, che la funzione built-in `print()` chiama, a sua volta, la funzione `str()`. Ciò significa che

la chiamata `print(obj)` comporta la chiamata a `obj.__str__()`. Nella prossima sezione, *L'ereditarietà come strumento per il riutilizzo del codice*, vedremo cosa accade quando il metodo `__str__()` non viene definito.

Vediamo brevemente come si comporta il nostro nuovo tipo di dato:

```
>>> p = Point()
>>> print(p)
(0, 0)
>>> p.distance()
0.0
>>> p = Point(3, 4)
>>> print(p)
(3, 4)
>>> p.distance()
5.0
```

Torniamo, quindi, alla nostra classe `Circle`, estendendola con l'aggiunta di un nuovo attributo di istanza, che rappresenta il centro del cerchio:

```
>>> class Circle:
...     def __init__(self, radius=1, center=Point()):
...         self.radius = radius
...         self.center = center
...     def area(self):
...         return math.pi * self.radius**2
...     def distance(self):
...         """Restituisci la distanza del centro dall'origine."""
...         return self.center.distance()
...     def __str__(self):
...         return '<radius: %f, center: %s>' %(self.radius, self.center)
...
>>> c = Circle()
>>> print(c)
<radius: 1.000000, center: (0, 0)>
>>> c = Circle(10, Point(3, 4))
>>> print(c)
<radius: 10.000000, center: (3, 4)>
>>> c.distance()
5.0
>>> c.area()
314.1592653589793
```

Ciò che abbiamo appena visto è un esempio di relazione di *composizione*, la quale tipicamente si manifesta quando un oggetto *ha un* (*has a*) altro oggetto.

In realtà, come abbiamo osservato prima, con Python questa relazione è naturale, perché ogni attributo fa riferimento a un oggetto, per cui si ha composizione sia nel caso del raggio sia in quello del centro.

L'unica differenza sta nel fatto che la relazione è più esplicita per il centro, poiché `Point` è un tipo di dato definito dall'utente.

L'ereditarietà

L'*ereditarietà* è il principale meccanismo di riutilizzo del codice nella OOP, grazie al quale è possibile far ereditare a una classe gli attributi di un'altra classe, in modo da non dover scrivere lo stesso codice più volte.

NOTA

La duplicazione del codice, nel medio e lungo periodo, conduce a un aumento dei tempi di mantenimento del software, visto che eventuali modifiche del codice vanno replicate. Come vedremo più avanti in questo capitolo, quando parleremo degli anti-pattern, l'ereditarietà è uno dei principali meccanismi di riutilizzo del codice che consente di evitare la diffusa pratica del *cut-and-paste programming*.

Per chiarire il concetto, consideriamo la seguente classe `Person`:

```
>>> import datetime
>>> class Person:
...     def __init__(self, name: str, surname: str, birthday: datetime.date):
...         self.name = name
...         self.surname = surname
...         self.birthday = birthday
...     def __str__(self):
...         return self.name + ' ' + self.surname
...
>>> max = Person('Max', 'Born', datetime.date(1882, 12, 11))
>>> max.name, max.birthday.year
('Max', 1882)
>>> max
<__main__.Person object at 0xb71d26ec>
>>> str(max)
'Max Born'
```

Supponiamo, adesso, di voler definire una classe che modella un tipo di dato che chiamiamo `Professor`, che rappresenta un generico professore:

```
>>> class Professor:
...     def __init__(self, name: str, surname: str, birthday: datetime.date, school: str):
```

```

... self.name = name
... self.surname = surname
... self.birthday = birthday
... self.school = school
... def __str__(self):
...     return self.name + ' ' + self.surname + ', at ' + self.school
...
>>> p = Professor('Max', 'Born', datetime.date(1882, 12, 11), ' University of Göttingen')
>>> str(p)
'Max Born, at University of Göttingen'

```

Come possiamo vedere, la classe `Professor` ha molto in comune con la classe `Person`. Questo è abbastanza normale, visto che un professore è *una* persona. Quando tra due classi vi è una relazione di questo tipo, chiamata in inglese *is-a* (è *un*), solitamente è una buona idea impiegare il meccanismo dell’ereditarietà, in modo da riutilizzare in una classe il codice scritto in un’altra classe. Nel nostro caso un professore è *una* persona (*a professor is a person*), per cui faremo in modo che la classe `Professor` erediti dalla classe `Person`. Se si vuole che una classe `B` erediti da una classe `A`, lo si deve indicare al momento della definizione di `B`, ponendo `A` tra parentesi, nel modo seguente:

```

>>> class A:
...     a = 33
...
>>> class B(A):
...     pass

```

Come possiamo vedere, la definizione della classe `B` è vuota. Nonostante ciò, poiché `B` eredita da `A`, tutti gli attributi definiti in `A` sono anche attributi di `B`:

```

>>> hasattr(B, 'a')
True
>>> B.a
33
>>> b = B()
>>> b.a
33

```

Quando una classe `B` eredita da una classe `A`, si dice che `A` è la *classe base* e `B` la *classe derivata*. Altri nomi alternativi sono *classe madre* o *superclasse* per `A` e *classe figlia* o *sottoclasse* per `B`.

Quindi, utilizzando il meccanismo dell’ereditarietà possiamo far ereditare la classe `Person` alla classe `Professor`, nel modo seguente:

```
>>> class Professor(Person):
...     def __init__(self, name: str, surname: str, birthday: datetime.date, school: str):
...         self.school = school
...     Person.__init__(self, name, surname, birthday)
...     def __str__(self):
...         return Person.__str__(self) + ', at ' + self.school
...
>>> max = Professor('Max', 'Born', datetime.date(1882, 12, 11), 'University of Göttingen')
>>> str(max)
'Max Born, at University of Göttingen'
>>> print(max.birthday)
1882-12-11
>>> Professor.__base__
<class '__main__.Person'>
```

In questa relazione di ereditarietà `Person` è la classe base, mentre `Professor` è quella derivata. Come abbiamo potuto vedere, l'attributo `__base__` di una classe è un riferimento alla sua classe base.

La classe built-in super

Nell'esempio precedente abbiamo fatto le chiamate `Person.__init__()` e `Person.__str__()`, qualificando i metodi con la classe base. Possiamo ottenere lo stesso risultato chiamando la classe built-in `super`:

```
>>> class Professor(Person):
...     def __init__(self, name: str, surname: str, birthday: datetime.date, school: str):
...         self.school = school
...         super(Professor, self).__init__(name, surname, birthday)
...     def __str__(self):
...         return super(Professor, self).__str__() + ', at ' + self.school
...
>>> david = Professor('David', 'Hilbert', datetime.date(1862, 1, 23),
... 'University of Göttingen')
>>> print(david)
David Hilbert, at University of Göttingen
>>> super # `super` è una classe built-in
<class 'super'>
```

La chiamata `super(Professor, self)` restituisce un oggetto che delega la chiamata dei metodi alla classe padre di `Professor`. Gli oggetti di questo tipo, che si comportano da intermediari, sono detti *proxy*. Il secondo parametro di `super` (`self`) viene passato al metodo come primo argomento, per cui, nel nostro caso,

`super(Professor, self).__str__()` è equivalente a `Person.__str__(self)`.

A partire da Python 3 (vedi PEP-3135), la chiamata alla classe `super` senza argomenti è equivalente alla chiamata `super(Professor, self)`:

```
>>> class Professor(Person):
...     def __init__(self, name: str, surname: str, birthday: datetime.date, school: str):
...         self.school = school
...     super().__init__(name, surname, birthday)
...     def __str__(self):
...         return super().__str__() + ', at ' + self.school
...
>>> david = Professor('David', 'Hilbert', datetime.date(1862, 1, 23),
... 'University of Göttingen')
>>> print(david)
David Hilbert, at University of Göttingen
```

Tipicamente si preferisce utilizzare quest'ultima modalità. Nelle prossime sezioni vedremo maggiori dettagli sul comportamento di `super`.

NOTA

La classe `super`, utilizzata senza argomenti, è una sorta di trucco magico. In fase di compilazione il parser verifica se si sta utilizzando l'etichetta `super` e in questo caso, se viene eseguita la chiamata senza argomenti, chiama `super` passandogli implicitamente `__class__` come primo argomento (che verrà poi valutato successivamente, a runtime) e l'istanza come secondo argomento. Quindi, se assegniamo `super` a un'altra etichetta al di fuori del metodo che esegue la chiamata, in fase di compilazione il parser non trova l'etichetta `super` e quindi non assegna gli argomenti in modo implicito; il codice, pertanto, non funziona:

```
>>> class A:
...     def mymethod(self, arg):
...         print('A.mymethod():', arg)
...
>>> mysuper = super
>>> class B(A):
...     def mymethod(self, arg):
...         mysuper().mymethod(arg)
...
>>> b = B()
>>> b.mymethod(100)
Traceback (most recent call last):
```

```
...
RuntimeError: super(): __class__ cell not found
```

Ovviamente tutto va a buon fine se passiamo gli argomenti in modo esplicito:

```
>>> mysuper = super
>>> class B(A):
...     def mymethod(self, arg):
...         mysuper(__class__, self).mymethod(arg)
...
>>> b = B()
>>> b.mymethod(100)
A.mymethod(): 100
```

Questo trucco magico è stato introdotto in Python 3 per questioni di praticità, poiché l'utilizzo di `super`, per i meno esperti, è spesso causa di confusione quando si tratta di passare esplicitamente gli argomenti. È una interessante interpretazione dello Zen di Python, che viola un principio (*esplicito è meglio che implicito*) per dare priorità a un altro (*la praticità batte la purezza*).

Ereditarietà multipla e ordine di risoluzione dei metodi

Python supporta l'*ereditarietà multipla*, ovvero una classe può ereditare da più classi:

```
>>> class A1:
...     a1 = 'python1'
...
>>> class A2:
...     a2 = 'python2'
...
>>> class A3:
...     a3 = 'python3'
...
>>> class B(A1, A2, A3):
...     pass
...
>>> b = B()
>>> b.a1, b.a2, b.a3
('python1', 'python2', 'python3')
```

Oltre all'attributo `__base__`, le classi hanno anche l'attributo `__bases__`, che è una tupla i cui elementi sono dei riferimenti alle classi base:

```
>>> B.__base__
<class '__main__.A1'>
>>> B.__bases__
(<class '__main__.A1'>, <class '__main__.A2'>, <class '__main__.A3'>)
```

Abbiamo visto come la classe `super` crei una istanza di un oggetto proxy, il quale delega le chiamate dei metodi alla classe base, evitando quindi di dover qualificare la classe in modo esplicito.

Però abbiamo appena visto che Python supporta l'ereditarietà multipla:

```
>>> class A1:
...     def foo(self):
...         print('A1.foo()')
...
>>> class A2:
...     def foo(self):
...         print('A2.foo()')
...
>>> class A3:
...     def foo(self):
...         print('A3.foo()')
...
>>> class B(A1, A2, A3):
...     def foo(self):
...         super().foo()
...
>>> b = B()
```

Se ora chiamiamo `b.foo()`, la chiamata `super().foo()` quale metodo chiama, quello di `A1`, di `A2` o di `A3`?

Vediamo un po':

```
>>> b.foo()
A1.foo()
```

L'ordine con cui vengono cercati gli attributi è chiamato *MRO* (*Method Resolution Order*) ed è indicato dall'attributo `__mro__` delle classi:

```
>>> B.__mro__
(<class '__main__.B'>, <class '__main__.A1'>, <class '__main__.A2'>, <class '__main__.A3'>, <class 'object'>)
```

Quando abbiamo chiamato `b.foo()`, questo ha chiamato `super().foo()`, ovvero `super(B, b).foo()`. Cosa fa, in pratica, `super(B, b).foo()`? Cerca la classe che nel MRO di `B` segue `B`, ovvero `A1`, e poi effettua la chiamata `A1.__foo__(b)`.

Se `A1` non avesse avuto il metodo `foo()`, la ricerca sarebbe continuata nella classe che, nel MRO di `B`, segue `A1`, ovvero `A2`. Se non fosse stato trovato in `A2`, e dopo neppure in `A3`, e poi neppure in `object`, allora la chiamata `b.foo()` avrebbe sollevato una eccezione di tipo `AttributeError`, mostrando come causa `'B' object has no attribute 'foo'`.

Nel MRO di `B` è presente la classe `object` poiché questa si trova in coda al MRO di tutte le classi:

```
>>> class A:
...     pass
...
>>> A.__mro__
(<class '__main__.A'>, <class 'object'>)
```

Come vedremo nel [Capitolo 6](#), questo è dovuto al fatto che la classe `object` si trova in cima alla gerarchia di ereditarietà, per cui ogni oggetto è istanza di `object`.

NOTA

Il MRO è un argomento avanzato. Se siamo interessati ad approfondirlo, possiamo leggere il seguente articolo di Guido <http://python-history.blogspot.it/2010/06/method-resolution-order.html>, oppure quello di Michele Simionato: <http://www.python.org/download/releases/2.3/mro/mro.txt>.

Nella prossima sezione parleremo ancora di MRO.

Usiamo `super`, perché è `super`

Il titolo di questa sezione è ispirato a quello di un articolo su `super`, pubblicato qualche anno fa da Raymond Hettinger: <http://rhettinger.wordpress.com/2011/05/26/super-considered-super/>. Il motivo più ovvio per usare `super` è che esso consente di evitare di riscrivere tutte le chiamate ai metodi delle classi base, nel caso in cui il nome di queste cambi. Consideriamo, ad esempio, il seguente codice:

```
>>> class A:
...     def foo(self):
...         print('python 3')
```

```
...
>>> class B(A):
...     def foo(self):
...         A.foo(self)
...     print("è meraviglioso :)")
...
>>> b = B()
>>> b.foo()
python 3
è meraviglioso :)
```

Se un giorno il nome della classe `A` cambia e diventa `Foo`, dobbiamo riscrivere tutte le chiamate `A.foo()`, sostituendole con `Foo.foo()`. Nel nostro esempio non è problema, ma potrebbe esserlo se la classe `A` implementa molti metodi ed è ereditata da molte classi. Quindi, meglio usare `super`:

```
>>> class B(A):
...     def foo(self):
...         super().foo()
...     print("è meraviglioso :)")
...
>>> b = B()
>>> b.foo()
python 3
è meraviglioso :)
```

In caso di ereditarietà multipla, `super` è quasi una necessità. Consideriamo, ad esempio, il seguente codice:

```
>>> class A:
...     def __init__(self):
...         print('A.__init__()')
...
>>> class A1(A):
...     def __init__(self):
...         A.__init__(self)
...         print('A1.__init__()')
...
>>> class A2(A):
...     def __init__(self):
...         A.__init__(self)
...         print('A2.__init__()')
...
>>> class B(A1, A2):
...     def __init__(self):
```

```

... A1.__init__(self)
... A2.__init__(self)
... print('B.__init__()')
...
>>> b = B()
A.__init__()
A1.__init__()
A.__init__()
A2.__init__()
B.__init__()

```

Come possiamo vedere, l'inizializzatore della classe base è stato chiamato due volte. Se, invece, deleghiamo tutte le chiamate a `super`, allora il metodo viene chiamato una sola volta:

```

>>> class A1(A):
...     def __init__(self):
...         super().__init__()
...         print('A1.__init__()')
...
>>> class A2(A):
...     def __init__(self):
...         super().__init__()
...         print('A2.__init__()')
...
>>> class B(A1, A2):
...     def __init__(self):
...         super().__init__()
...         print('B.__init__()')
...
>>> b = B()
A.__init__()
A2.__init__()
A1.__init__()
B.__init__()

```

Per capire ciò che è successo, iniziamo col dare uno sguardo al MRO di `B`:

```

>>> B.__mro__
(<class '__main__.B'>, <class '__main__.A1'>, <class '__main__.A2'>, <class '__main__.A'>, <class 'object'>)

```

Ora seguiamo il flusso di esecuzione, partendo dalla chiamata `b = B()`. Ecco cosa è avvenuto:

1. l'istruzione `b = B()` ha causato la chiamata `b.__init__()` (in realtà l'etichetta `b` non è ancora stata assegnata, ma dal punto di vista logico non cambia nulla);

2. la prima istruzione di `b.__init__()` è `super().__init__()`, ovvero `super(B, b).__init__()`. Se guardiamo il MRO, la classe che segue B è A1, per cui `super(B, b).__init__()` viene tradotta in `A1.__init__(b)`;
3. la prima istruzione eseguita da `A1.__init__(b)` è `super().__init__()`, ovvero `super(A1, b).__init__()`. La classe che segue A1 nel MRO di B è A2, per cui `super(A1, b).__init__()` viene tradotta in `A2.__init__(b)`;
4. la prima istruzione eseguita da `A2.__init__(b)` è, ancora una volta, `super().__init__()`, la quale corrisponde a `super(A2, b).__init__()`. La classe che segue A2 nel MRO di B è A, per cui la chiamata `super(A2, b).__init__()` viene tradotta in `A.__init__(b)`;
5. `A.__init__(b)` ha stampato la stringa '`A.__init__()`' e poi ha restituito il controllo a `A2.__init__()`;
6. `A2.__init__()` ha stampato la stringa '`A2.__init__()`' e poi ha restituito il controllo a `A1.__init__()`;
7. `A1.__init__()` ha stampato la stringa '`A1.__init__()`' e poi ha restituito il controllo a `B.__init__()`;
8. `B.__init__()` ha stampato la stringa '`B.__init__()`' e poi ha restituito il controllo al suo chiamante (nel [Capitolo 6](#) scopriremo chi è il chiamante di `__init__()`), il quale ha assegnato l'istanza all'etichetta `b`.

Quindi, in definitiva, usiamo `super` e deleghiamo a lui le chiamate ai metodi della classe base, perché `super` sa come muoversi, è una *super* classe!

Overriding e overwriting dei metodi delle classi base

Consideriamo la seguente classe A:

```
>>> class A:
...     def foo(self):
...         print('A.foo()')
```

Se una classe B che eredita da A non definisce il metodo `B.foo()`, allora, come sappiamo, lo eredita da A:

```
>>> class B(A):
...     pass
...
>>> b = B()
>>> b.foo()
A.foo()
```

Se lo definisce, ma si limita a chiamare `A.foo()`, allora la situazione è identica alla

precedente:

```
>>> class B(A):
...     def foo(self):
...         super().foo()
...
>>> b = B()
>>> b.foo()
A.foo()
```

Se, invece, `B.foo()`, oltre a chiamare il metodo della classe base, esegue anche dell'altro codice, come mostrato, ad esempio, di seguito:

```
>>> class B(A):
...     def foo(self):
...         super().foo() # Chiamo il metodo della classe base
...         print('B.foo()') # Codice specifico della classe derivata
...
>>> b = B()
>>> b.foo()
A.foo()
B.foo()
```

allora si dice che `B` fa l'*overriding* del metodo `foo()`. In altre parole, `B` *estende* il metodo `foo()` della classe base.

Se, invece, la classe `B` non chiama il metodo `foo()` della classe base, allora si dice che fa l'*overwriting* di `foo()`, come ad esempio nel caso seguente:

```
>>> class B(A):
...     def foo(self):
...         print('B.foo()')
...
>>> b = B()
>>> b.foo()
B.foo()
```

Quindi fare l'*overwriting* di un metodo significa sovrascriverlo (rimpiazzarlo), mentre farne l'*overriding* significa estenderlo. L'*overwriting* serve, quindi, per rimpiazzare un comportamento, mentre l'*overriding* per estenderlo.

Introduzione alle classi base astratte e al modulo `abc`

Supponiamo di voler definire una classe `PlaneShape` che rappresenti una generica figura geometrica, caratterizzata dall'avere un'area e un perimetro.

Ovviamente, per una generica figura geometrica, non siamo in grado di implementare i metodi `area()` e `perimeter()`, visto che non vi è una formula per calcolarli: l'area del cerchio si calcola in un modo, quella del rettangolo in un altro, quella del triangolo in un altro ancora, e così via. Questo non è un problema, perché non vogliamo implementare questi metodi in `PlaneShape`. Vogliamo, invece, che `PlaneShape` definisca una interfaccia generica, e che siano le classi che ereditano da essa a implementare i metodi `area()` e `perimeter()`. Una soluzione potrebbe essere la seguente:

```
>>> class PlaneShape:
...     """Figura geometrica piana"""
...     def area(self):
...         raise NotImplementedError("Il metodo non è stato ancora implementato")
...     def perimeter(self):
...         raise NotImplementedError("Il metodo non è stato ancora implementato")
```

Se proviamo a chiamare un metodo di `PlaneShape`, ci viene detto che non è stato ancora implementato:

```
>>> s = PlaneShape()
>>> s.area()
Traceback (most recent call last):
...
NotImplementedError: Il metodo non è stato ancora implementato
```

Ora tutte le figure geometriche che ereditano da `PlaneShape` hanno una interfaccia comune, ovvero hanno i metodi `area()` e `perimeter()`:

```
>>> class Rectangle(PlaneShape):
...     pass
...
>>> hasattr(Rectangle(), 'area')
True
>>> hasattr(Rectangle(), 'perimeter')
True
```

Le classi derivate, però, devono fare l'overwriting di questi metodi, altrimenti, se vengono chiamati, viene sollevata una eccezione:

```
>>> square = Rectangle()
>>> square.area()
Traceback (most recent call last):
...
NotImplementedError: Il metodo non è stato ancora implementato
```

Vediamo qualche classe che eredita da `PlaneShape` e implementa i due metodi. Questo è un esempio di rettangolo:

```
>>> class Rectangle(PlaneShape):
...     """Figura geometrica piana con angoli interni di 90 gradi"""
...     def __init__(self, *sides):
...         self.sides = sides
...         self.length = max(sides)
...         self.width = min(sides)
...     def area(self):
...         return self.length * self.width
...     def perimeter(self):
...         return sum(self.sides)
...
>>> square = Rectangle(10, 10, 10, 10)
>>> square.perimeter()
40
>>> square.area()
100
```

Questo, invece, è un esempio di cerchio:

```
>>> import math
>>> class Circle(PlaneShape):
...     """Cerchio"""
...     def __init__(self, radius):
...         self.radius = radius
...     def area(self):
...         return math.pi * self.radius**2
...     def perimeter(self):
...         return 2 * math.pi * self.radius
...
>>> c = Circle(10)
>>> c.area()
314.1592653589793
>>> c.perimeter()
62.83185307179586
```

NOTA

Dovremmo verificare che il valore dei lati del rettangolo e quello del raggio del cerchio siano non negativi, ma per gli scopi di questa sezione può andare

bene così.

Una classe di questo tipo, che definisce una interfaccia ma non la implementa completamente, è detta *classe base astratta*. I metodi definiti ma non implementati sono detti *metodi astratti*.

L'ideale è che sia le classi base astratte sia quelle che ereditano da esse e non implementano i metodi astratti delle classi base non siano istanziabili. In questo modo non si va incontro al rischio di chiamare un metodo non implementato. Il modulo `abc` della libreria standard ci consente di ottenere questo comportamento. Per fare ciò dobbiamo definire `PlaneShape` nel seguente modo:

```
>>> class PlaneShape(metaclass=ABCMeta):
...     """Figura geometrica piana"""
...     def area(self):
...         """Restituisci l'area della figura geometrica"""
...         area = abstractmethod(area)
...
...     def perimeter(self):
...         """Restituisci il perimetro della figura geometrica"""
...         perimeter = abstractmethod(perimeter)
...
```

I metodi sono stati resi astratti con l'istruzione `metodo = abstractmethod(metodo)`.

NOTA

Tra breve vedremo che l'assegnamento `metodo = abstractmethod(metodo)` si può fare in modo conciso utilizzando la sintassi dei decoratori, che consiste nell'inserire, subito prima della definizione dei metodi, una linea contenente `@abstractmethod`:

```
>>> class PlaneShape(metaclass=ABCMeta):
...     @abstractmethod
...     def area(self):
...         pass
...     @abstractmethod
...     def perimeter(self):
...         pass
```

Ora non siamo più in grado di istanziare `PlaneShape`:

```
>>> PlaneShape()
Traceback (most recent call last):
...
TypeError: Can't instantiate abstract class PlaneShape with abstract methods area, perimeter
```

Inoltre, se una classe eredita da `PlaneShape` ma non definisce tutti i metodi astratti, allora non può essere istanziata:

```
>>> class Rectangle(PlaneShape):
...     """Figura geometrica piana con angoli interni di 90 gradi"""
...     def __init__(self, *sides):
...         self.sides = sides
...         self.length = max(sides)
...         self.width = min(sides)
...     def area(self):
...         return self.length * self.width
...
>>> Rectangle()
Traceback (most recent call last):
...
TypeError: Can't instantiate abstract class Rectangle with abstract methods perimeter
```

Comportamento di `isinstance()` e `issubclass()` con l'ereditarietà

Consideriamo tre classi, una che rappresenta una generica figura geometrica, un'altra che rappresenta un quadrilatero e la terza un quadrato, e supponiamo che siano legate tramite una relazione di ereditarietà:

```
>>> class Shape:
...     """Figura geometrica"""
...
>>> class Rectangle(Shape):
...     """Rettangolo: figura geometrica con angoli interni di 90 gradi"""
...
>>> class Square(Rectangle):
...     """Quadrato: quadrilatero con angoli interni di 90 gradi e lati uguali"""
```

Abbiamo visto che l'attributo `__bases__` di una classe è una tupla contenente le sue classi base, per cui potremmo pensare di utilizzarlo per verificare se una classe è derivata da un'altra:

```
>>> Rectangle in Square.__bases__
```

True

Con questo test, però, non possiamo risalire la gerarchia di ereditarietà:

```
>>> Shape in Square.__bases__  
False
```

Potremmo pensare di farlo verificando l'appartenenza di Shape a Square.__mro__:

```
>>> Shape in Square.__mro__  
True
```

Il modo corretto di effettuare questo test è farlo in modo esplicito, utilizzando una apposita funzione built-in chiamata `issubclass()`:

```
>>> issubclass(Square, Rectangle)  
True  
>>> issubclass(Square, Shape)  
True
```

Si osservi che una classe è considerata sottoclasse di se stessa:

```
>>> issubclass(Square, Square)  
True
```

NOTA

Se una classe `B` viene registrata come sottoclasse di una classe astratta `A`, allora la classe `A` non è presente nel MRO di `B`, mentre `issubclass(B, A)` restituisce `True`:

```
>>> class A(metaclass=ABCMeta):  
...     pass  
...  
>>> class B:  
...     pass  
...  
>>> A.register(B)  
<class '__main__.B'>  
>>> issubclass(B, A)  
True  
>>> A in B.__mro__  
False
```

La funzione `isinstance()` va invece utilizzata tutte le volte che vogliamo sapere se un oggetto è istanza di una data classe. Infatti sia l'attributo `__class__` sia la classe built-in `type` non risalgono la gerarchia di ereditarietà, mentre `isinstance()` sì:

```
>>> Square().__class__ is Shape
False
>>> type(Square()) is Shape
False
>>> isinstance(Square(), Shape)
True
>>> isinstance(Square(), Rectangle)
True
```

I decoratori

Se ricordiamo, nella sezione sulle classi base astratte abbiamo definito la classe `PlaneShape`:

```
>>> from abc import ABCMeta, abstractmethod
>>> class PlaneShape(metaclass=ABCMeta):
...     """Figura geometrica piana"""
...     def area(self):
...         """Restituisci l'area della figura geometrica"""
...         area = abstractmethod(area)
...
...     def perimeter(self):
...         """Restituisci il perimetro della figura geometrica"""
...         perimeter = abstractmethod(perimeter)
```

Sostanzialmente abbiamo utilizzato la funzione `abc.abstractmethod()` per *decorare* dei metodi, ovvero le abbiamo passato il metodo, lei ha compiuto delle azioni su di esso in modo da modificarne il comportamento, poi lo ha restituito in una versione modificata che noi abbiamo riassegnato alla medesima etichetta. In questo caso le etichette prima e dopo la decorazione fanno riferimento al medesimo oggetto:

```
>>> class PlaneShape(metaclass=ABCMeta):
...     def area(self):
...         pass
...         print(id(area))
...         area = abstractmethod(area)
...         print(id(area))
...     def perimeter(self):
...         pass
...         print(id(perimeter))
...         area = abstractmethod(area)
...         print(id(perimeter))
...
140288077727648
140288077727648
140288077727792
```

Possiamo chiarire ancora meglio il concetto di *decoratore* con un esempio. Consideriamo la seguente funzione:

```
>>> def mydecorator(func):
...     func.lang = 'python'
...     return func
```

Questa prende un oggetto come argomento, lo *decora*, ovvero gli aggiunge qualcosa (l'attributo `lang`), e poi lo restituisce. Possiamo usare questa funzione come *decoratore* in modo identico a quanto visto con `abstractmethod`. Ad esempio, definiamo una funzione `foo()`:

```
>>> def foo():
...     pass
...
>>> id(foo)
3072326364
>>> hasattr(foo, 'lang')
False
```

Se la passiamo a `mydecorator()` e poi assegniamo l'oggetto restituito nuovamente all'etichetta `foo`, il risultato è che `foo` ha un nuovo attributo *iniettato* da `mydecorator()`:

```
>>> foo = mydecorator(foo)
>>> id(foo)
3072326364
>>> hasattr(foo, 'lang')
True
>>> foo.lang
'python'
```

Questa operazione è detta *decorazione*, proprio per il fatto che ha l'effetto di *decorare* un oggetto con nuove o diverse funzionalità. Una funzione o una classe utilizzata per decorare un oggetto è detta *decoratore* (*decorator*). In particolare, quando l'oggetto da decorare è una funzione, come nel nostro caso, questo processo è detto *decorazione di funzione* e il decoratore è detto *decoratore di funzione*. Analogamente, quando l'oggetto da decorare è una classe, il processo è detto *decorazione di classe* e il decoratore è detto *decoratore di classe*.

In Python esiste una sintassi apposita per realizzare in modo conciso una

decorazione. Infatti, dato un decoratore `mydecorator`, se si vuole decorare con esso una funzione o una classe, è sufficiente scrivere `@mydecorator` nella linea che precede la definizione della funzione o della classe:

```
>>> @mydecorator
... def foo():
...     pass
...
>>> hasattr(foo, 'lang')
True
>>> foo.lang
'python'
```

Quindi, tornando alla definizione della classe base astratta, il seguente esempio:

```
>>> class PlaneShape(metaclass=ABCMeta):
...     def area(self):
...         pass
...     area = abstractmethod(area)
...     def perimeter(self):
...         pass
...     area = abstractmethod(area)
```

viene usualmente scritto in modo conciso utilizzando la sintassi dei decoratori:

```
>>> class PlaneShape(metaclass=ABCMeta):
...     @abstractmethod
...     def area(self):
...         pass
...     @abstractmethod
...     def perimeter(self):
...         pass
```

I decoratori possono essere applicati anche in cascata. In questo caso vanno indicati ciascuno in una linea diversa:

```
>>> def mydecorator1(obj):
...     obj.lang = 'python'
...     return obj
...
>>> def mydecorator2(obj):
```

```

... obj.version = '3.x'
... return obj
...
>>> @mydecorator1
... @mydecorator2
... def foo():
...     pass
...
>>> foo.lang
'python'
>>> foo.version
'3.x'

```

La funzione `functools.wraps()`

Nella sezione precedente abbiamo visto un esempio di decoratore che restituisce la stessa funzione presa come argomento. In realtà questo accade di rado; tipicamente, un decoratore restituisce una funzione differente da quella che prende come argomento ed effettua la chiamata alla funzione da decorare all'interno di quest'ultima. Ad esempio, nel seguente caso la funzione da decorare viene chiamata all'interno di `mywrapper()` e poi il decoratore restituisce `mywrapper()`:

```

>>> def mydecorator(f):
...     def mywrapper(*args, **kwargs):
...         """Wrapper della funzione da decorare"""
...         print('Eseguo delle azioni prima di chiamare', f.__name__)
...         result = f(*args, **kwargs)
...         print('Eseguo altre azioni dopo aver chiamato', f.__name__)
...         return result
...     return mywrapper

```

La funzione `mywrapper()` è una sorta di involucro della funzione originale, nel senso che copre la funzione originale con del codice aggiuntivo. Le funzioni di questo tipo sono chiamate *wrapper*. Ecco cosa accade quando decoriamo una funzione con `mydecorator()`:

```

>>> @mydecorator
... def foo(a, b):
...     """Restituisci a + b"""
...     return a + b
...
>>> foo(1, 2)
Eseguo delle azioni prima di chiamare foo

```

Eseguo altre azioni dopo aver chiamato foo

3

Questa implementazione del decoratore ha il grave inconveniente di sovrascrivere i metadati della funzione originale, come, ad esempio, il nome e la docstring:

```
>>> foo.__name__  
'mywrapper'  
>>> foo.__doc__  
'Wrapper della funzione da decorare'
```

Il problema si risolve decorando il wrapper con la funzione `functools.wraps()` della libreria standard:

```
>>> import functools  
>>> def mydecorator(f):  
...     @functools.wraps(f)  
...     def mywrapper(*args, **kwargs):  
...         """Wrapper della funzione da decorare"""  
...         print('Eseguo delle azioni prima di chiamare', f.__name__)  
...         result = f(*args, **kwargs)  
...         print('Eseguo altre azioni dopo aver chiamato', f.__name__)  
...         return result  
...     return mywrapper  
...  
>>> @mydecorator  
... def foo(a, b):  
...     """Restituisci a + b"""  
...     return a + b  
...  
>>> foo(1, 2)  
Eseguo delle azioni prima di chiamare foo  
Eseguo altre azioni dopo aver chiamato foo  
3  
>>> foo.__name__  
'foo'  
>>> foo.__doc__  
"""Restituisci a + b"
```

L'attributo `__wrapped__` di `foo()` fa riferimento alla funzione originale (quella *wrappata*):

```
>>> f = foo.__wrapped__  
>>> f(1, 2)  
3
```

Decoratori che accettano argomenti

Abbiamo detto che la sintassi dei decorator consente di scrivere questo codice:

```
def foo():  
    pass  
foo = myfunction(foo)
```

nel seguente modo:

```
@myfunction  
def foo():  
    pass
```

Il seguente codice, invece:

```
def foo():  
    pass  
foo = myfunction(x, y)(foo)
```

è equivalente a questo:

```
@myfunction(x, y)  
def foo():  
    pass
```

In questo caso la chiamata `myfunction(x, y)` deve restituire un oggetto callable che prende almeno un argomento, in modo che questo oggetto venga a sua volta chiamato passandogli la funzione `foo()`. Consideriamo, ad esempio, il seguente codice:

```
>>> import functools  
>>> def externalfun(x, y, z):  
...     def mydecorator(f):  
...         @functools.wraps(f)  
...         def mywrapper(*args, **kwargs):  
...             print('In mywrapper:', (x, y, z))  
...             return f(*args, **kwargs)  
...         return mywrapper  
...     return mydecorator
```

Ecco cosa accade quando decoriamo una funzione con `externalfun()` e poi la chiamiamo:

```
>>> @externalfun(1, 2, 3)
... def foo(a, b):
...     return a + b
...
>>> foo('a', 'b')
In mywrapper: (1, 2, 3)
'ab'
```

Vediamo di capire cosa è successo, seguendo il flusso di esecuzione passo dopo passo. Come abbiamo detto, il codice precedente alla chiamata `foo('a', 'b')` è equivalente al seguente:

```
>>> def foo(a, b):
...     return a + b
...
>>> foo = externalfun(1, 2, 3)(foo)
```

Con la chiamata `externalfun(1, 2, 3)` viene definita la funzione `mydecorator()`, che poi viene restituita. Quindi la seguente:

```
foo = externalfun(1, 2, 3)(foo)
```

è equivalente a:

```
foo = mydecorator(foo)
```

Il vantaggio di aver definito `mydecorator()` all'interno di `externalfun()` è che gli argomenti passati a quest'ultima sono visibili a `mydecorator()` e precisamente si trovano nel suo scope enclosed, per cui, come abbiamo visto nel [Capitolo 4](#), sono tenuti in memoria anche se `externalfun()` ha terminato la sua esecuzione.

Facciamo ancora un altro passo, perché il risultato della chiamata `mydecorator(foo)` è la definizione e restituzione di una nuova funzione, `mywrapper()`, per cui la precente istruzione è equivalente a questa:

```
foo = mywrapper
```

Quindi, in definitiva, la chiamata `foo('a', 'b')` non è altro che la chiamata `mywrapper('a', 'b')`, la quale stampa gli argomenti passati a `externalfun()`, chiama la funzione originale `foo()` (che vede nel suo scope enclosed) passandole 'a' e 'b', e restituisce

il risultato di questa chiamata, ovvero 'a' + 'b'.

I metodi e le property

I metodi possono appartenere a tre categorie: *metodi di classe*, *metodi statici* e *metodi di istanza*. Ai metodi di classe viene passato implicitamente come primo argomento la classe, mentre a quelli di istanza viene passata l'istanza. Questi due tipi di metodo sono quindi legati alla classe o all'istanza e questo è il motivo per cui sono detti *bound methods*. Solitamente i metodi operano sulle istanze, per cui i metodi di istanza sono quelli con cui si ha a che fare nella maggior parte dei casi. Al termine di questa sezione mostreremo qualche caso d'uso, in modo da chiarire ulteriormente le differenze fra i tre tipi di metodo e illustrare dei possibili ambiti di utilizzo dei metodi statici e di classe.

Metodi di classe

I *metodi di classe* sono dei metodi ai quali viene implicitamente passata la classe come primo argomento. Se vogliamo che un metodo `foo()` sia di classe, dobbiamo renderlo tale decorandolo con la classe built-in `classmethod`, la quale lo modifica rendendolo, per l'appunto, di classe:

```
>>> class Foo:
...     @ classmethod
...     def foo(cls):
...         print(id(cls))
...
>>> id(Foo)
3072370444
>>> Foo.foo() # La classe 'Foo' viene passata in modo implicito al metodo
3072370444
```

NOTA

Il primo parametro del metodo di classe viene convenzionalmente chiamato `cls`, come indicato nella sezione *Function and method arguments* della PEP-0008.

Un metodo di classe è tale a prescindere da come viene qualificato, nel senso che gli viene passata implicitamente la classe anche se lo qualificiamo tramite l'istanza:

```
>>> f = Foo()
>>> f.foo()
3072370444
```

La classe viene passata in ogni caso, anche se non definiamo il parametro `cls`:

```
>>> class Foo:
...     @ classmethod
...     def foo():
...         print('foo')
...
>>> Foo.foo()
Traceback (most recent call last):
...
TypeError: foo() takes 0 positional arguments but 1 was given
```

I metodi di classe sono oggetti di tipo `types.MethodType`:

```
>>> type(Foo.foo)
<class 'method'>
>>> import types
>>> type(Foo.foo) is types.MethodType
True
```

Metodi statici

I metodi non di classe che vengono qualificati tramite la classe sono detti *metodi statici* (*static methods*). A questi metodi non viene passato implicitamente alcun argomento, per cui sostanzialmente sono delle semplici funzioni che vivono nello scope locale della classe.

```
>>> class Foo:
...     def foo():
...         print('python')
...
>>> f = Foo()
>>> type(Foo.foo), type(f.foo)
(<class 'function'>, <class 'method'>)
>>> Foo.foo() # Nessun argomento viene passato in modo implicito
python
>>> import types
>>> type(Foo.foo) is types.FunctionType
True
```

È possibile rendere un metodo statico in modo esplicito, analogamente a quanto è stato fatto con i metodi di classe, decorandolo con la classe built-in `staticmethod`.


```
>>> class Foo:
...     @staticmethod
...     def foo():
...         print('python')
```

Quando un metodo è dichiarato statico esplicitamente, allora è statico a prescindere da come viene qualificato:

```
>>> f = Foo()
>>> type(Foo.foo), type(f.foo) # È statico anche se viene qualificato con l'istanza
(<class 'function'>, <class 'function'>)
>>> Foo.foo()
python
>>> f.foo()
python
```

NOTA

In Python 2 è possibile definire un metodo statico solo in modo esplicito:

```
>>> class Foo:
...     def foo():
...         print('foo')
...
>>> Foo.foo()
Traceback (most recent call last):
...
TypeError: unbound method foo() must be called with Foo instance as first argument (got nothing instead)
>>> class Foo:
...     @staticmethod
...     def foo():
...         print('foo')
...
>>> Foo.foo()
foo
```

Metodi di istanza

Quando un metodo viene qualificato tramite l'istanza, se non è di classe o statico, allora l'istanza gli viene passata implicitamente come primo argomento.

I metodi di questo tipo vengono detti *metodi di istanza* e sono oggetti di tipo `types.MethodType`:

```
>>> class Foo:
```

```
... def foo(self):
...     print(self)
...
>>> f = Foo()
>>> type(f.foo)
<class 'method'>
>>> f.foo() # L'istanza viene passata implicitamente come primo argomento
<__main__.Foo object at 0xb7205d6c>
```

Se un metodo di istanza non definisce il parametro `self`, gli viene comunque passata l'istanza implicitamente come primo parametro:

```
>>> class Foo:
...     def foo():
...         print('foo')
...
>>> f = Foo()
>>> f.foo()
Traceback (most recent call last):
...
TypeError: foo() takes 0 positional arguments but 1 was given
```

Differenze fra i tre tipi di metodo

Come abbiamo già detto, poiché ai metodi di classe e di istanza viene sempre passato implicitamente un argomento (rispettivamente, la classe e l'istanza a cui sono legati), vengono detti *bound methods*:

```
>>> class Foo:
...     @classmethod
...     def foo():
...         pass
...
>>> Foo.foo # Un metodo di classe è un bound method
<bound method type.foo of <class '__main__.Foo'>>
>>> class Foo:
...     def foo():
...         pass
...
>>> f = Foo()
>>> f.foo # Un metodo di istanza è un bound method
<bound method Foo.foo of <__main__.Foo object at 0xb720cb4c>>
```

Abbiamo visto che i metodi di classe e di istanza sono oggetti di tipo `types.MethodType`, mentre quelli statici sono delle semplici funzioni, ovvero oggetti di tipo `types.FunctionType`:

```
>>> class Foo:
...     @classmethod
...     def cmethod(cls):
...         print(cls)
...     @staticmethod
...     def smethod():
...         print('smethod')
...     def imethod(self):
...         print(self)
...
>>> f = Foo()
>>> type(f.cmethod), type(f.imethod), type(f.smethod)
(<class 'method'>, <class 'method'>, <class 'function'>)
```

I metodi di istanza e di classe, a differenza dei metodi statici, hanno gli attributi `__self__` e `__func__`:

```
>>> set(dir(f.imethod)) - set(dir(f.cmethod))
set()
>>> set(dir(f.imethod)) - set(dir(f.smethod))
{'__func__', '__self__'}
```

L'attributo `__self__` fa riferimento all'oggetto passato implicitamente come primo argomento, quindi la classe per un metodo di classe e l'istanza per un metodo di istanza:

```
>>> f.cmethod.__self__ is Foo, f.imethod.__self__ is f
(True, True)
```

L'attributo `__func__` di un bound method fa riferimento alla versione statica del metodo:

```
>>> type(f.cmethod.__func__), type(f.imethod.__func__)
(<class 'function'>, <class 'function'>)
>>> f.cmethod.__func__('abc') # La classe non viene passata implicitamente
abc
>>> f.imethod.__func__('def') # L'istanza non viene passata implicitamente
def
>>> f.imethod.__func__ is Foo.imethod
True
```

Concludiamo riassumendo le differenze fra i tre tipi di metodo. Per default un metodo qualificato tramite la classe è un metodo statico, ovvero una semplice funzione che non riceve implicitamente alcun argomento:

```
>>> class Foo:
...     def foo(obj):
...         print(obj)
...
>>> Foo.foo(44)
44
```

Un metodo qualificato tramite l'istanza è per default un metodo di istanza, ovvero un metodo al quale viene implicitamente passata l'istanza come primo argomento:

```
>>> class Foo:
...     def foo(self, obj):
...         print(id(self), obj, sep='--')
...
>>> f = Foo()
>>> id(f)
140246682605456
>>> f.foo(44)
140246682605456--44
```

Questi comportamenti di default cambiano quando si effettua una *decorazione* dei metodi con le funzioni built-in `staticmethod()` e `classmethod()`. La prima rende il metodo statico, indipendentemente da come viene qualificato:

```
>>> class Foo:
...     @staticmethod
...     def foo(obj):
...         print(obj)
...
>>> f = Foo()
>>> Foo.foo(44)
44
>>> f.foo(44)
44
```

La seconda rende il metodo di classe, per cui gli viene passata implicitamente la classe come primo argomento, indipendentemente da come viene qualificato:

```
>>> class Foo:
...     @classmethod
...     def foo(cls):
...         print(cls)
...
```

```
>>> f = Foo()
>>> Foo.foo()
<class '__main__.Foo'>
>>> f.foo()
<class '__main__.Foo'>
```

Casi d'uso

Come abbiamo già detto, tipicamente si vuole che i metodi agiscano sulle istanze della classe, per cui nella maggior parte dei casi avremo a che fare con metodi di istanza. L'ambito di utilizzo dei metodi di classe e dei metodi statici è quindi più raro e specialistico; in questa sezione vedremo qualche caso d'uso, in modo da farci un'idea di come e quando usarli.

Metodi di classe per inizializzare in differenti modi le istanze

Consideriamo il seguente file *words.py*:

```
$ cat words.py
class Words:
    def __init__(self, words):
        self.words = tuple(word.strip() for word in words)
```

```
@classmethod
def fromfile(cls, file_name):
    return cls(open(file_name))
```

```
@classmethod
def fromstr(cls, string, sep=' '):
    return cls(string.split(sep))
def __str__(self):
    return 'Words' + str(self.words)
```

Questo definisce una classe `Words` che sostanzialmente rappresenta un contenitore di parole. Le istanze della classe possono essere inizializzate in tre modi diversi. Il primo modo è quello classico, che, come sappiamo, consiste nel chiamare direttamente la classe. In questo caso si passa a `Words` un oggetto iterabile i cui elementi sono stringhe, e questo oggetto viene convertito in tupla (dopo aver fatto uno `str.strip()` delle stringhe):

```
>>> from words import Words
>>> w = Words(['cane', 'gatto', 'mucca', 'cavallo'])
>>> print(w)
Words('cane', 'gatto', 'mucca', 'cavallo')
>>> d = {'cane': 'bau', 'gatto': 'miao', 'pecora': 'beee'}
>>> w = Words(d)
>>> print(w)
Words('gatto', 'cane', 'pecora')
```

Se vogliamo inizializzare l'istanza mediante una stringa di parole, come ad esempio 'mare montagna', non dobbiamo chiamare la classe passandole la stringa, perché, così facendo, otterremo un contenitore di caratteri:

```
>>> w = Words('mare montagna')
>>> print(w)
Words('m', 'a', 'r', 'e', ' ', 'm', 'o', 'n', 't', 'a', 'g', 'n', 'a')
```

Dobbiamo utilizzare, invece, il metodo di classe `Words.fromstr()`, che non è altro che un iniziatore alternativo:

```
>>> w = Words.fromstr('mare montagna pianura collina')
>>> print(w)
Words('mare', 'montagna', 'pianura', 'collina')
>>> w = Words.fromstr('mare; montagna; pianura; collina', sep=';')
>>> print(w)
Words('mare', 'montagna', 'pianura', 'collina')
```

Questo metodo prende la stringa e da questa ottiene una lista chiamando `str.split()`. Nel nostro caso la lista risultante è:

```
>>> 'mare montagna pianura collina'.split()
['mare', 'montagna', 'pianura', 'collina']
```

Questa è stata passata come argomento a `cls`, che, come sappiamo, è un riferimento alla classe `Words`, visto che `Words.fromstr()` è un metodo di classe. Infine, il metodo `Words.fromstr()` ha restituito `Words(['mare', 'montagna', 'pianura', 'collina'])`, ovvero una nuova istanza inizializzata nel modo classico. Abbiamo definito `Words.fromstr()` come metodo di classe perché ci serve per creare e inizializzare una istanza, per cui ci occorre operare sulla classe e non sull'istanza, che ancora non esiste. Avremmo potuto definirlo anche statico, ma, se avessimo fatto questa scelta, avremmo dovuto passare esplicitamente la classe come primo argomento:

```
>>> w = Words.fromstr(Words, 'mare montagna pianura collina')
```

Infine, possiamo inizializzare l'istanza chiamando il metodo di classe `Words.fromfile()`, passandogli il nome di un file. Questo deve contenere una parola per linea:

```
$ cat myfile
mela
pera
```

```

ciliegia
banana
$ python -c "from words import Words; print(Words.fromfile('myfile'))"
Words('mela', 'pera', 'ciliegia', 'banana')

```

Chiudiamo questa sezione mostrando un esempio un po' più completo di classe `Words`:

```

$ cat words.py
class Words:
    def __init__(self, words):
        self.words = tuple(word.strip() for word in words)
    for word in self.words:
        if not word.isalpha():
            raise ValueError("%s: parola non valida" %word)

```

```

@classmethod
def fromfile(cls, file_name):
    return cls(open(file_name))

```

```

@classmethod
def fromstr(cls, string, sep=' '):
    return cls(string.split(sep))
def __str__(self):
    return self.__class__.__name__ + str(self.words)

```

Ora il metodo `Words.__init__()` verifica che le parole siano composte da soli caratteri dell'alfabeto:

```

>>> from words import Words
>>> w = Words(['mela', 'pera', 'banana'])
>>> w = Words(['mela1', 'pera', 'banana'])
Traceback (most recent call last):
...
ValueError: mela1: parola non valida

```

NOTA

Se definiamo `Words` in modo che `Words.__str__()` restituisca `'Words' + str(self.words)`, allora, quando ereditiamo dalla classe `Words`, andiamo incontro a una sorpresa. Consideriamo, ad esempio, il seguente caso:

```

$ cat words.py
class Words:
    def __init__(self, words):
        self.words = tuple(word.strip() for word in words)

```

```

@classmethod

```

```
def fromfile(cls, file_name):
    return cls(open(file_name))

@classmethod
def fromstr(cls, string, sep=' '):
    return cls(string.split(sep))
def __str__(self):
    return 'Words' + str(self.words)
```

```
class Foo(Words):
    pass
```

Ecco cosa accade se proviamo a stampare una istanza di `Foo`:

```
>>> from words import Foo
>>> f = Foo(['cane', 'gatto', 'mucca', 'cavallo'])
>>> print(f)
Words('cane', 'gatto', 'mucca', 'cavallo')
```

Vogliamo, invece, che venga stampato `Foo('cane', 'gatto', 'mucca', 'cavallo')`, per cui, piuttosto che indicare il nome della classe `Words`, dobbiamo ottenere il nome della classe dinamicamente, usando `self.__class__.__name__`, come mostrato di seguito:

```
$ cat words.py
class Words:
    def __init__(self, words):
        self.words = tuple(word.strip() for word in words)
```

```
@classmethod
def fromfile(cls, file_name):
    return cls(open(file_name))
```

```
@classmethod
def fromstr(cls, string, sep=' '):
    return cls(string.split(sep))
def __str__(self):
    return self.__class__.__name__ + str(self.words)
```

```
class Foo(Words):
    pass
```

Come possiamo vedere, adesso tutto funziona correttamente:

```
>>> from words import Words, Foo
>>> f = Foo(['cane', 'gatto', 'mucca', 'cavallo'])
>>> print(f)
Foo('cane', 'gatto', 'mucca', 'cavallo')
>>> w = Words(['cane', 'gatto', 'mucca', 'cavallo'])
```



```
>>> print(w)
Words('cane', 'gatto', 'mucca', 'cavallo')
```

Esempio di utilizzo di un metodo statico

Se dobbiamo compiere varie azioni, ma non dobbiamo operare sulla classe stessa o sull'istanza, allora possiamo usare un metodo statico. Consideriamo, ad esempio, la classe `Words` appena vista e supponiamo di voler raccogliere all'interno di una funzione sia l'operazione `str.strip()` delle parole sia la loro validazione. Abbiamo due alternative: o creiamo una funzione esterna alla classe, come nel seguente esempio:

```
$ cat words.py
def sanitize(words):
    words = tuple(word.strip() for word in words)
    for word in words:
        if not word.isalpha():
            raise ValueError("%s: parola non valida" %word)
    return words
```

```
class Words:
    def __init__(self, words):
        self.words = sanitize(words)
```

```
@classmethod
def fromfile(cls, file_name):
    return cls(open(file_name))
```

```
@classmethod
def fromstr(cls, string, sep=' '):
    return cls(string.split(sep))
```

```
def __str__(self):
    return self.__class__.__name__ + str(self.words)
```

oppure, visto che queste operazioni, nonostante non agiscano sulla classe, sono comunque legate dal punto di vista logico alla classe, creiamo un metodo statico:

```
$ cat words.py
class Words:
    def __init__(self, words):
        self.words = Words.sanitize(words)
```

```
@classmethod
def fromfile(cls, file_name):
    return cls(open(file_name))
```

```

@classmethod
def fromstr(cls, string, sep=' '):
    return cls(string.split(sep))

def __str__(self):
    return self.__class__.__name__ + str(self.words)

```

```

@staticmethod
def sanitize(words):
    words = tuple(word.strip() for word in words)
    for word in words:
        if not word.isalpha():
            raise ValueError("%s: parola non valida" %word)
    return words

```

Concludiamo questa sezione introducendo due metodi speciali, `__iter__()` e `__contains__()`, che vedremo meglio nel [Capitolo 6](#):

```

$ cat words.py
class Words:
    def __init__(self, words):
        self.words = Words.sanitize(words)

```

```

@classmethod
def fromfile(cls, file_name):
    return cls(open(file_name))

```

```

@classmethod
def fromstr(cls, string, sep=' '):
    return cls(string.split(sep))

```

```

def __str__(self):
    return self.__class__.__name__ + str(self.words)

```

```

def __iter__(self):
    return iter(self.words)

```

```

def __contains__(self, word):
    return True if word in self.words else False

```

```

@staticmethod
def sanitize(words):
    words = tuple(word.strip() for word in words)
    for word in words:
        if not word.isalpha():
            raise ValueError("%s: parola non valida" %word)
    return words

```

Il metodo `Word.__iter__()` restituisce un iteratore, per cui consente alle istanze di `Words` di essere oggetti iterabili:

```
>>> for word in Words(['mela', 'pera', 'banana']):
print(word)
...
mela
pera
banana
```

Il metodo `Words.__contains__()` consente di effettuare dei test di appartenenza:

```
>>> w = Words(['mela', 'pera', 'banana'])
>>> 'mela' in w
True
>>> 'albicocca' in w
False
```

Questi metodi hanno un significato speciale per Python. Come abbiamo detto, ne parleremo in modo dettagliato nella sezione *Gli attributi magici* del Capitolo 6.

Le property

Una *property* è un attributo che possiamo considerare speciale, poiché è definita come un metodo, nonostante non sia un oggetto *callable*. Consideriamo la seguente classe `Circle` e concentriamoci solo sul metodo `Circle.area()`:

```
>>> import math
>>> class Circle:
...     def __init__(self, r=1):
...         self.radius = r
...     def area(self):
...         return math.pi * self.radius**2
```

Se usiamo la classe built-in `property` per decorare `Circle.area()`, otteniamo un attributo `Circle.area` di tipo `property`, il quale non è un oggetto *callable*, ma nonostante ciò ogni volta che viene referenziato tramite l'istanza esegue il codice della funzione e restituisce il valore, esattamente come se fosse stato chiamato il metodo di istanza:

```
>>> class Circle:
...     def __init__(self, r=1):
...         self.radius = r
```

```
... @property
... def area(self):
...     return math.pi * self.radius**2
...
>>> type(Circle.area)
<class 'property'>
>>> callable(Circle.area)
False
>>> c = Circle()
>>> c.area # È come se avessimo eseguito `c.area()`
3.141592653589793
>>> type(c.area) # È come se avessimo eseguito `type(c.area())`
<class 'float'>
```

Parleremo ancora delle property nel Capitolo 6, nella sezione dedicata ai *descriptor*.

Introduzione ai design pattern

La sezione *Casi d'uso* ci consente di introdurre un importante argomento, quello dei *design pattern*. Partiamo dalla considerazione che un dato problema può essere risolto in vari modi, come abbiamo visto nella sezione *Esempio di utilizzo di un metodo statico*. Inoltre, molti problemi sono comuni a vari programmatori e un programmatore, tipicamente, incontra i problemi comuni varie volte nell'arco della sua attività professionale.

Per i problemi che ci capita frequentemente di dover risolvere, l'ideale sarebbe poterne archiviare da qualche parte la soluzione, in modo da recuperarla e riutilizzarla all'occorrenza successiva, evitando così di reinventare la ruota ogni volta che ci ritroviamo ad affrontarlo. Sarebbe, inoltre, molto utile potersi confrontare con altri programmatori, specialmente se hanno già affrontato e risolto il problema, in modo da capire qual è la soluzione migliore, o per lo meno i vantaggi e gli svantaggi delle varie soluzioni messe a confronto. Come possiamo immaginare, più il problema è complesso, più una soluzione condivisa e di comprovata efficacia ci farà risparmiare tempo in futuro. Siamo, così, arrivati al dunque: i *design pattern*.

Il movimento pattern

Nel 1995 Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, soprannominati *la banda dei quattro* (Gang of Four - GoF), pubblicarono un libro intitolato *Design Patterns: Elements of Reusable Object-Oriented Software*, che introduceva il concetto di design pattern nella programmazione. Il libro ebbe un tale successo in ambito informatico da portare alla nascita di un'intera corrente di pensiero a favore della progettazione basata sui pattern, detta, per l'appunto, il *movimento pattern*.

Nella parte introduttiva del libro, precisamente nella sezione *What is a Design Pattern?*, la GoF riporta la seguente citazione dall'architetto Christopher Alexander, che rappresenta la prima descrizione del concetto di design pattern:

Ogni pattern descrive un problema che si presenta frequentemente nel nostro ambiente, e quindi descrive il nucleo della soluzione così che si possa impiegare tale soluzione milioni di volte, senza peraltro produrre due volte la stessa realizzazione.

In poche parole, un design pattern descrive un problema comune e la relativa

soluzione. In generale, un pattern dovrebbe avere quattro elementi essenziali:

1. *nome*: un *nome* sintetico (una o due parole) che identifica il design pattern in modo unico;
2. *descrizione*: la descrizione del *problema* che il design pattern intende affrontare;
3. *soluzione*: la descrizione di una *soluzione* generica al problema che il pattern risolve;
4. *conseguenze*: la spiegazione delle *conseguenze* che l'applicazione della soluzione proposta può comportare.

Una volta che il pattern è stato ben caratterizzato sulla base dei suoi quattro elementi essenziali e accettato da una comunità di sviluppatori, diventa anche un ottimo strumento per documentare il software, visto che basta specificare il nome del pattern applicato per capire immediatamente ciò che fa il codice. I pattern, quindi, consentono di utilizzare un vocabolario comune agli sviluppatori, in modo da facilitare la comunicazione tra di loro.

La classificazione dei design pattern

I pattern, sulla base del loro scopo, vengono usualmente classificati in tre categorie:

- *creazionali*: riguardano il processo di creazione degli oggetti;
- *strutturali*: riguardano la struttura dati degli oggetti;
- *comportamentali*: riguardano le funzionalità degli oggetti.

Passiamo alla pratica e vediamo due esempi di design pattern creazionali.

Monostate

In questo esempio ci occupiamo di un pattern chiamato *monostate* (singolo stato), o anche *borg*, il quale affronta il seguente problema: si vuole che le istanze di una classe condividano lo stato.

Soluzione

Nel linguaggio comune, quando parliamo di stato di un oggetto intendiamo la condizione in cui l'oggetto si trova. Diciamo, ad esempio, che un oggetto si trova in buono stato o in cattivo stato. Lo stato, in generale, cambia nel tempo, per cui un oggetto che oggi è in buono stato domani potrebbe essere in cattivo stato.

Lo stato tipicamente è determinato sulla base dei valori di alcune proprietà

dell'oggetto. Ad esempio, per un'autovettura lo stato può essere determinato sulla base delle condizioni in cui si trovano il motore, gli pneumatici, la carrozzeria e il sistema frenante. Secondo questo modello, il motore, la carrozzeria e il sistema frenante rappresentano le proprietà dell'autovettura, e lo stato dell'auto è determinato dalla condizione in cui si trovano queste sue proprietà.

In modo analogo, dal punto di vista software, lo *stato di una istanza* è determinato dall'insieme degli attributi di quella istanza e dagli oggetti a cui tali attributi fanno riferimento. Anche lo stato di una istanza, quindi, in generale, cambia nel tempo. Vediamo di capire con un esempio ciò di cui stiamo parlando. Consideriamo, ancora una volta, la classe `Person`:

```
>>> import datetime
>>> class Person:
...     def __init__(self, name, surname: str, birthday: datetime.date):
...         self.name = name
...         self.surname = surname
...         self.birthday = birthday
...     def __str__(self):
...         return self.name + ' ' + self.surname
...
...
>>> max = Person('Max', 'Born', datetime.date(1882, 12, 11))
>>> david = Person('David', 'Hilbert', datetime.date(1862, 1, 23))
```

Come abbiamo detto, lo stato di una istanza è determinato dai suoi attributi e dagli oggetti ai quali tali attributi fanno riferimento. In questo caso, gli unici attributi che identificano lo stato sono il nome, il cognome e il compleanno della persona. I restanti attributi, infatti, sono comuni a tutte le istanze.

In questo momento le due istanze hanno, pertanto, due stati differenti, visto che `max.name` fa riferimento a una stringa di valore `'Max'`, `max.surname` a una stringa `'Born'` e `max.birthday` a `datetime.date(1862, 1, 23)`, mentre `david.name` fa riferimento alla stringa `'David'`, `david.surname` a `'Hilbert'` e `david.birthday` a `datetime.date(1862, 1, 23)`.

Se riflettiamo un attimo, in Python lo stato di una istanza coincide con il namespace:

```
>>> max.__dict__
{'name': 'Max', 'surname': 'Born', 'birthday': datetime.date(1882, 12, 11)}
>>> david.__dict__
{'name': 'David', 'surname': 'Hilbert', 'birthday': datetime.date(1862, 1, 23)}
```

Ecco, quindi, la soluzione al nostro problema:

```
>>> class Borg:
...     _shared_state = {}
...     def __init__(self):
...         self.__dict__ = self._shared_state
...
```

Infatti l'assegnamento `self.__dict__ = self._shared_state` fa sì che il namespace di ogni istanza sia l'attributo di classe a cui fa riferimento `_shared_state`, che, essendo di classe, è per l'appunto condiviso da tutte le istanze:

```
>>> a = Borg()
>>> b = Borg()
>>> a.__dict__
{}
>>> b.__dict__
{}
>>> a.__dict__ is b.__dict__
True
>>> a.foo = 44
>>> a.__dict__
{'foo': 44}
>>> b.__dict__
{'foo': 44}
```

Se vogliamo ereditare dalla classe `Borg`, *prima* di inizializzare le istanze della classe derivata dobbiamo chiamare l'inizializzatore della classe base:

```
>>> class Foo(Borg):
...     def __init__(self, value):
...         super().__init__() # Come prima cosa chiamiamo l'inizializzatore di Borg
...         self.value = value
```

In questo modo anche il namespace delle istanze di `Foo` fa riferimento a `Borg._shared_state`, per cui tutte le istanze di `Borg` condividono lo stesso namespace:

```
>>> c = Foo('python')
>>> isinstance(c, Borg)
True
>>> c.__dict__
{'foo': 44, 'value': 'python'}
>>> a.__dict__
{'foo': 44, 'value': 'python'}
>>> d = Foo([1, 2, 3])
>>> d.__dict__
{'foo': 44, 'value': [1, 2, 3]}
```



```
>>> c.__dict__
{'foo': 44, 'value': [1, 2, 3]}
>>> a.__dict__
{'foo': 44, 'value': [1, 2, 3]}
```

Osserviamo come le istanze, nonostante abbiano lo stesso stato, siano oggetti diversi:

```
>>> a is b
False
>>> a is c
False
>>> c is d
False
```

NOTA

Il borg, in realtà, non è un vero e proprio pattern, visto che il problema non è ricorrente e per giunta è molto semplice da risolvere. In questi casi, quindi, al termine *pattern* si preferisce il termine *idioma*, per cui diremo *monostate idiom* piuttosto che *monostate pattern*. Nonostante non sia un vero e proprio pattern, dal punto di vista didattico è esattamente equivalente a un vero pattern.

Nel Capitolo 6, quando parleremo della creazione delle istanze, vedremo un altro pattern creazionale imparentato con il borg, chiamato *singleton pattern*. Il singleton ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, per cui, come ovvio effetto secondario, si ha che tutte le istanze di tale classe hanno lo stesso stato.

Gli anti-pattern

Spesso si sente parlare anche di *anti-pattern*. Questo termine è stato utilizzato per la prima volta da Andrew Koenig nel suo articolo “Patterns and Antipatterns”, pubblicato nel 1995 dal *Journal of Object-Oriented Programming*.

Gli anti-pattern descrivono soluzioni comunemente utilizzate nel processo di sviluppo software che però conducono a conseguenze decisamente negative, le quali, a prima vista, sfuggono ai più. Descrivono, quindi, una soluzione comunemente utilizzata, spiegano il motivo per cui, in realtà, non è una soluzione valida e mostrano come porre rimedio. Sono descritti da:

- un *nome* sintetico (una o due parole) che identifica il pattern in modo

unico;

- la *forma* in cui si manifesta;
- i *sintomi* che consentono di riconoscere l'anti-pattern;
- le *cause* che portano a adottare l'anti-pattern;
- la *soluzione* che descrive come eliminare l'anti-pattern.

Passiamo alla pratica e vediamo un esempio concreto di anti-pattern: il *cut-and-paste programming*.

Cut-and-paste programming

Il *cut-and-paste programming* (programmazione copia-e-incolla) non ha bisogno di presentazioni. Di per sé, questa pratica non è nemica del programmatore, ma può diventarlo se utilizzata da persone senza esperienza o con scarse conoscenze di programmazione. Queste, infatti, la utilizzano per cercare delle soluzioni già pronte, o anche delle soluzioni parziali che possono poi usare come base di partenza per i loro programmi, non sentendosi in grado di scrivere del codice interamente di mano propria.

Forma

Tipicamente, lavorano al progetto dei programmatori alle prime armi, che tendono a copiare, ed eventualmente modificare, codice già scritto da colleghi che reputano più esperti. Nel progetto sono quindi presenti varie parti di codice simili o addirittura identiche.

Sintomi

Questi programmatori generalmente non comprendono totalmente il codice copiato, per cui, anche se nel breve periodo questo approccio porta a risparmiare del tempo, nel medio-lungo termine conduce a una serie di problemi che annullano i vantaggi iniziali:

- lo stesso bug si ripete nonostante siano già state fatte molte correzioni;
- alti costi di manutenzione del software e duplicazione degli errori;
- difficoltà nel localizzare le porzioni di codice che conducono al medesimo errore.

Cause

Le cause che generalmente portano a utilizzare questa pratica sono le seguenti:

- creare del codice da zero è faticoso e si preferisce avere un ritorno d'investimento immediato anziché a lungo termine;

- la velocità di sviluppo del codice è più importante della qualità del codice stesso;
- gli sviluppatori hanno limitate conoscenze teoriche e non sono in grado di riutilizzare il codice dei colleghi per mezzo dell'ereditarietà e della composizione;
- gli sviluppatori non pensano a quello che verrà dopo.

Soluzione

Per risolvere il problema:

- i programmatori devono sviluppare il software tenendo a mente che deve risultare facile poterlo riutilizzare tramite il meccanismo dell'ereditarietà;
- il team di lavoro deve sapere che la pratica del copia e incolla è assolutamente da evitare e che il riutilizzo del codice va fatto impiegando le strategie appropriate, come ad esempio l'ereditarietà;
- i programmatori devono sempre verificare la possibilità di riutilizzare, con i metodi appropriati, codice già prodotto;
- il progetto nella sua interezza (codice, tools e framework utilizzati) deve essere ben documentato.

Le eccezioni

Come abbiamo detto nel [Capitolo 1](#), gli errori in un programma Python si manifestano sotto forma di *errori di sintassi* oppure di *eccezioni*. Gli errori di sintassi vengono rilevati durante la fase di compilazione del programma in bytecode, quindi prima che il programma venga eseguito, mentre le eccezioni vengono sollevate a runtime. Consideriamo, ad esempio, il seguente file:

```
$ cat foo.py
def foo()
print('foo')
prin(__name__)
foo()
```

Come possiamo vedere, abbiamo dimenticato di scrivere i due punti nell'istruzione `def`. Questo è un errore di sintassi e quindi verrà segnalato prima che il programma vada in esecuzione. Infatti, come possiamo verificare, la stringa `'foo'` non viene stampata:

```
$ python -c "import foo"
Traceback (most recent call last):
File "<string>", line 1, in <module>
File "./foo.py", line 1
def foo()
^
SyntaxError: invalid syntax
```

Quando nel codice vi è un errore di sintassi, il bytecode ovviamente non viene generato:

```
$ ls # Il bytecode non è stato generato
foo.py
```

Ora sistemiamo le cose, correggendo l'errore di sintassi:

```
$ cat foo.py
def foo():
print('foo')
prin(__name__)
foo()
```

Adesso il programma è sintatticamente corretto, ma abbiamo commesso un

errore di battitura, scrivendo `prin()` anziché `print()`. Questo errore può essere rilevato solo a runtime, quando la funzione `foo()` viene eseguita, perché, come abbiamo visto nel [Capitolo 4](#), solo in quel momento Python cercherà di risolvere l'etichetta libera. Infatti, come possiamo vedere, viene generato il bytecode ed eseguito il programma sino alla stampa della stringa `'foo'`:

```
$ python -c "import foo"
foo
Traceback (most recent call last):
...
NameError: global name 'prin' is not defined
$ ls # Il bytecode è stato generato
foo.py __pycache__
```

Quando l'esecuzione del programma arriva all'istruzione `prin(__name__)`, Python cerca di risolvere l'etichetta libera `prin`, ma, non trovando alcuna definizione di questa, solleva una eccezione.

Come abbiamo già detto nel [Capitolo 1](#), e come possiamo notare da questo esempio, quando le eccezioni non vengono gestite dal programma, l'esecuzione termina e viene mostrato un messaggio di errore. Tale messaggio si differenzia da quelli dovuti agli errori di sintassi principalmente per la presenza del *traceback*.

Il messaggio di errore

Il messaggio di errore che viene mostrato quando il programma termina a seguito di una eccezione è composto da due parti: il *traceback* e l'indicazione del *tipo di eccezione* e della *causa dell'errore*.

Il traceback

Il termine *traceback* deriva dalla composizione delle due parole inglesi, *trace* e *back*. Infatti il traceback tiene traccia (*trace*) di tutte le linee di codice coinvolte nell'errore, partendo dalla meno recente sino ad arrivare alla più recente, ovvero all'indietro (*back*). Il traceback è un oggetto e ciò che viene indicato come *Traceback* nel messaggio di errore non è altro che la stampa dell'oggetto traceback.

Per capire cosa indica la stampa del traceback, consideriamo il seguente script:

```
$ cat myscript.py
def foo():
    mylist = [1, 2, 3]
    item = mylist[5] # Solleva una eccezione
```

```
def moo():  
    foo()
```

```
moo()
```

Se lo eseguiamo, il programma solleva una eccezione:

```
$ python myscript.py  
Traceback (most recent call last):  
File "myscript.py", line 8, in <module>  
    moo()  
File "myscript.py", line 6, in moo  
    foo()  
File "myscript.py", line 3, in foo  
    item = mylist[5] # Solleva una eccezione  
IndexError: list index out of range
```

Come possiamo vedere, ogni traccia mostrata nella stampa del traceback consiste in due linee. La prima riporta il nome del file contenente la linea di codice coinvolta, il numero della linea e lo scope, mentre la seconda è la linea di codice. Ad esempio, nel nostro caso la prima traccia è:

```
File "myscript.py", line 8, in <module>  
moo()
```

Questa dice che nel file *myscript.py*, alla linea numero 8 nello scope al top level del modulo (<module>), è stata eseguita la linea di codice `moo()`. La seconda traccia è la seguente:

```
File "myscript.py", line 6, in moo  
foo()
```

Questa dice che nel file *myscript.py*, alla linea numero 6, nel corpo della funzione `moo()`, è stata eseguita la linea di codice `foo()`. Infine, l'ultima traccia è la seguente:

```
File "myscript.py", line 3, in foo  
item = mylist[5] # Solleva una eccezione
```

Questa dice che nel file *myscript.py*, alla linea numero 3, nel corpo della funzione `foo()`, è stata eseguita la linea di codice `item = mylist[5]`. Essendo questa l'ultima linea eseguita, sappiamo che la causa dell'errore è da ricercare qui. Per agevolare la diagnosi, il messaggio di errore nell'ultima linea riporta il tipo e la causa dell'errore. Nel nostro caso ci viene detto che abbiamo indicizzato

la lista con un indice fuori dal range di valori consentiti:

IndexError: list index out of range

NOTA

Quando lavoriamo in modo interattivo, non viene riportata la linea di codice e quindi le tracce sono composte, ciascuna, da una linea:

```
>>> def moo():
...     foo()
...
>>> def foo():
...     mylist = [1, 2, 3]
...     item = mylist[5] # Solleva una eccezione
...
>>> moo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in moo
  File "<stdin>", line 3, in foo
IndexError: list index out of range
```

Inoltre, come possiamo osservare, per ciascuna traccia il conteggio dei numeri di linea inizia a partire dal prompt (`>>>`) della relativa linea di codice.

Vediamo anche un esempio nel quale sono coinvolti più file. Definiamo nel file *foo.py* la funzione `foo()` che solleva l'eccezione, e nel file *moo.py* la funzione `moo()` che chiama `foo()`:

```
$ cat moo.py
from foo import foo
def moo():
    foo()
    moo()
$ cat foo.py
def foo():
    mylist = [1, 2, 3]
    item = mylist[5] # Solleva una eccezione
```

Ora eseguiamo *moo.py*:

```
$ python moo.py
Traceback (most recent call last):
  File "moo.py", line 5, in <module>
```

```
moo()
File "moo.py", line 3, in moo
foo()
File "/home/marco/temp/foo.py", line 3, in foo
item = mylist[5] # Solleva una eccezione
IndexError: list index out of range
```

Come possiamo vedere, le prime due tracce riguardano il file *moo.py*, mentre l'ultima il file *foo.py*.

Il tipo di eccezione

Nei vari esempi di codice esaminati in questo libro abbiamo incontrato diversi messaggi di errore. Avrete sicuramente notato che tutti terminano con l'indicazione del tipo di eccezione che è stata sollevata, seguita dalla causa dell'errore. Si parla di *tipo* di eccezione a proposito, perché le eccezioni sono istanze di particolari classi. Quindi, ad esempio, `IndexError`, `NameError` e `TypeError` sono delle classi:

```
>>> IndexError
<class 'IndexError'>
>>> NameError
<class 'NameError'>
>>> TypeError
<class 'TypeError'>
```

Queste classi sono particolari nel senso che sono sottoclassi della classe built-in `BaseException`. Approfondiremo il discorso tra breve.

La causa dell'errore

Le eccezioni hanno un attributo chiamato `args`:

```
>>> ex = IndexError()
>>> ex.args
()
```

Come possiamo vedere, questo è una tupla e contiene gli argomenti passati alla classe al momento della creazione dell'istanza:

```
>>> ex = IndexError("Descrizione della causa dell'errore")
>>> ex.args
("Descrizione della causa dell'errore",)
>>> ex = IndexError([1, 2, 3], 'python')
>>> ex.args
([1, 2, 3], 'python')
```


Quando stampiamo una eccezione, la funzione `print()` verifica la lunghezza di `args`. Se è pari a uno, allora stampa l'unico elemento di `args`:

```
>>> ex = IndexError("Messaggio di errore...")
>>> print(ex)
Messaggio di errore...
```

Se invece `args` contiene più di un elemento, allora `print()` stampa la tupla intera:

```
>>> ex = IndexError([1, 2, 3], 'python')
>>> print(ex)
([1, 2, 3], 'python')
```

Nella maggior parte dei casi, le eccezioni vengono create passando alla classe una stringa che descrive la causa dell'errore (quindi un solo argomento). Questa causa è ciò che viene mostrato nell'ultima linea del messaggio di errore, subito dopo i due punti che seguono il tipo di eccezione:

```
>>> mylist = [1, 2, 3]
>>> mylist[5]
Traceback (most recent call last):
...
IndexError: list index out of range
```

In questo caso, ad esempio, è stata sollevata l'eccezione `IndexError('list index out of range')`, quindi la causa dell'errore è descritta come *list index out of range*.

La gestione delle eccezioni

La gestione delle eccezioni, alla quale abbiamo accennato nel [Capitolo 1](#), avviene tramite l'istruzione `try` e la sua clausola `except`. In questa sezione affronteremo l'argomento più in dettaglio, per cui parleremo anche delle clausole `else` e `finally` e mostreremo un esempio nel quale metteremo in pratica i concetti appresi.

La clausola except

Consideriamo il seguente esempio:

```
>>> mylist = ['a', 'b', 'c']
>>> while True:
...     i = int(input("Inserisci un numero intero: "))
...     print(mylist[i])
...
Inserisci un numero intero: 0
```

```
a
Inserisci un numero intero: 1
b
Inserisci un numero intero:
```

Se inseriamo un valore dell'indice fuori dal range ammissibile, l'istruzione `print(mylist[i])` solleva una eccezione di tipo `IndexError` e il programma termina la sua esecuzione mostrando un messaggio di errore:

```
Inserisci un numero intero: 4
Traceback (most recent call last):
...
IndexError: list index out of range
```

Vediamo cosa accade se inseriamo l'istruzione `print(mylist[i])` nella suite di una clausola `try` e gestiamo l'eventuale eccezione di tipo `IndexError` con la clausola `except`:

```
>>> mylist = ['a', 'b', 'c']
>>> while True:
...     try:
...         i = int(input("Inserisci un numero intero: "))
...         print(mylist[i])
...         print('Siamo nella suite della try')
...     except IndexError:
...         print('Siamo nella suite della except')
...         print('Siamo al di fuori del blocco try/except')
...
Inserisci un numero intero: 1
b
Siamo nella suite della try
Siamo al di fuori del blocco try/except
```

Come possiamo vedere, se nella suite della `try` non viene sollevata l'eccezione, allora dopo l'ultima istruzione della suite della `try` il programma salta la suite della `except` e continua la sua normale esecuzione. Se, invece, nella suite della `try` un'istruzione solleva un'eccezione, l'esecuzione da quella linea passa direttamente alla clausola `except`, la quale, se gestisce quel tipo di eccezione, esegue la sua suite:

```
Inserisci un numero intero: 5
Siamo nella suite della except
Siamo al di fuori del blocco try/except
```

Quindi, dopo che la suite della `except` è stata eseguita, il programma (se non vengono sollevate delle eccezioni nella suite stessa) continua il suo normale funzionamento riprendendo l'esecuzione dalla prima istruzione che segue il blocco `try/except`.

Se all'interno della suite della `try` viene sollevata una eccezione di un tipo diverso da `IndexError`, la `except` non può gestirla, per cui la sua suite non viene eseguita e il programma termina:

```
Inserisci un numero intero: due
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'due'
```

Questa eccezione è stata sollevata dalla funzione built-in `int()`, la quale non riesce a convertire la stringa `'due'` in un numero intero.

Gestire diversi tipi di eccezione nella stessa clausola `except`

Come abbiamo visto nel [Capitolo 1](#), se vogliamo far gestire a una clausola `except` diversi tipi di eccezione, possiamo elencarli in una tupla, nel modo seguente:

```
>>> mylist = ['a', 'b', 'c']
>>> while True:
...     try:
...         i = int(input("Inserisci un numero intero: "))
...         print(mylist[i])
...         print('Siamo nella suite della try')
...     except (IndexError, ValueError):
...         print('Siamo nella suite della except')
...         print('Siamo al di fuori del blocco try/except')
...
Inserisci un numero intero: due
Siamo nella suite della except
Siamo al di fuori del blocco try/except
Inserisci un numero intero: 5
Siamo nella suite della except
Siamo al di fuori del blocco try/except
```

Gestire le eccezioni in più clausole `except`

In alternativa, possiamo utilizzare più clausole `except`, in modo da gestire in maniera differente ciascun tipo di eccezione:

```
>>> mylist = ['a', 'b', 'c']
>>> while True:
...     try:
...         i = int(input("Inserisci un numero intero: "))
...         print(mylist[i])
...         print('Siamo nella suite della try')
...     except IndexError:
...         print('Indice non valido!')
...     except ValueError:
...         print('Non riesco a convertire in numero il testo inserito.')
...     print('Siamo al di fuori del blocco try/except')
...
Inserisci un numero intero: due
Non riesco a convertire in numero il testo inserito.
Siamo al di fuori del blocco try/except
Inserisci un numero intero: 5
Indice non valido!
Siamo al di fuori del blocco try/except
```

Se più clausole `except` gestiscono il medesimo tipo di eccezione, solamente la suite della prima di esse verrà eseguita:

```
>>> mylist = ['a', 'b', 'c']
>>> while True:
...     try:
...         i = int(input("Inserisci un numero intero: "))
...         print(mylist[i])
...         print('Siamo nella suite della try')
...     except IndexError:
...         print('Indice non valido!')
...     except IndexError:
...         print('Seconda suite che gestisce IndexError...')
...     except ValueError:
...         print('Non riesco a convertire in numero il testo inserito.')
...     print('Siamo al di fuori del blocco try/except')
...
Inserisci un numero intero: 5
Indice non valido!
Siamo al di fuori del blocco try/except
```

La forma `except as` consente di avere le informazioni sull'eccezione

L'istruzione `except` in combinazione con la parola chiave `as` consente di assegnare l'eccezione a una etichetta. Ad esempio, la clausola `except ExceptionType as`

`exception` assegna l'eccezione all'etichetta `exception`. Possiamo, quindi, utilizzare questa forma per stampare la causa dell'errore:

```
>>> mylist = ['a', 'b', 'c']
>>> while True:
...     try:
...         i = int(input("Inserisci un numero intero: "))
...         print(mylist[i])
...         print('Siamo nella suite della try')
...     except IndexError as ex:
...         print('Errore:', ex)
...     except ValueError as ex:
...         print('Errore:', ex)
...     print('Siamo al di fuori del blocco try/except')
...
Inserisci un numero intero: 5
Errore: list index out of range
Siamo al di fuori del blocco try/except
Inserisci un numero intero: due
Errore: invalid literal for int() with base 10: 'due'
Siamo al di fuori del blocco try/except
```

Come possiamo vedere, abbiamo assegnato l'eccezione all'etichetta `ex`, e poi l'abbiamo stampata nella suite della `except`.

Catturare tutti i tipi di eccezione con una unica clausola `except` generica

Se nella clausola `except` non viene indicato alcun tipo di eccezione da gestire, allora vengono catturate tutte le eccezioni:

```
>>> mylist = ['a', 'b', 'c']
>>> while True:
...     try:
...         i = int(input("Inserisci un numero intero: "))
...         print(mylist[i])
...         print('Siamo nella suite della try')
...     except:
...         print("È successo qualcosa di strano...")
...         print('Il programma continua la sua normale esecuzione')
...
Inserisci un numero intero: due
È successo qualcosa di strano...
Il programma continua la sua normale esecuzione
Inserisci un numero intero: 5
```

È successo qualcosa di strano...
Il programma continua la sua normale esecuzione

Come abbiamo già detto nel Capitolo 1, utilizzare `except` in questo modo, per pura pigrizia, generalmente porta a dover impiegare molto più tempo di quello che si è guadagnato.

NOTA

Si osservi che in questo caso non riusciamo a uscire dal ciclo `while` neppure digitando una interruzione da tastiera (**Control-C**), perché la clausola `except` generica cattura ogni tipo di eccezione e quindi anche le eccezioni di tipo `KeyboardInterrupt`.

Tra breve vedremo che in Python 3 tutti i tipi di eccezione sono sottoclassi di `BaseException` e che la clausola `except Exception` è da preferire alla `except` generica. In ogni caso, se si utilizza una `except` generica bisogna quanto meno avere cura di mostrare il tipo e la causa dell'errore. Per fare ciò si utilizza la funzione `sys.exc_info()`.

Ottenere informazioni sull'eccezione catturata con una `except` generica

Vi sono vari modi per ottenere informazioni sull'ultima eccezione catturata con una `except` generica. Uno di questi consiste nell'utilizzo della funzione `sys.exc_info()`, la quale restituisce una tupla di tre elementi, contenente il *tipo* dell'eccezione catturata, l'eccezione stessa e il *traceback*. Ecco un esempio di utilizzo di `sys.exc_info()`:

```
>>> import sys
>>> mylist = ['a', 'b', 'c']
>>> while True:
...     try:
...         i = int(input("Inserisci un numero intero: "))
...         print(mylist[i])
...         print('Siamo nella suite della try')
...     except IndexError:
...         print('Indice non valido!')
...     except ValueError:
...         print('Non riesco a convertire in numero il testo inserito.')
...     except:
```

```
... print('\nSiamo nella suite della except generica')
... print('Tipo di eccezione:', sys.exc_info()[0])
... print('Siamo al di fuori del blocco try/except', end='\n\n')
...
Inserisci un numero intero: 5
Indice non valido!
Siamo al di fuori del blocco try/except
```

```
Inserisci un numero intero: due
Non riesco a convertire in numero il testo inserito.
Siamo al di fuori del blocco try/except
```

```
Inserisci un numero intero:
Siamo nella suite della except generica
Tipo di eccezione: <class 'KeyboardInterrupt'>
Siamo al di fuori del blocco try/except
```

Nell'ultimo prompt abbiamo inserito una interruzione da tastiera e, come possiamo vedere, è stata sollevata una eccezione; questa è stata catturata dalla `except generica`, la quale ha stampato `sys.exc_info()[0]`, ovvero il tipo dell'eccezione catturata.

Ovviamente solo nella suite della `except` possiamo ottenere le informazioni sull'eccezione catturata, per cui, al di fuori di questa, i tre elementi della tupla sono `None`:

```
>>> import sys
>>> sys.exc_info()
(None, None, None)
```

Un altro modo per ottenere informazioni sull'errore consiste nell'utilizzare le funzioni del modulo `traceback` della libreria standard. Ad esempio, per stampare il messaggio di errore senza interrompere il programma si utilizza la funzione `traceback.print_exc()`:

```
$ cat foo.py
import traceback
mytuple = (1, 2, 3)
try:
    print(mytuple[5]) # Solleva una eccezione
except:
    traceback.print_exc() # Stampa il messaggio di errore

print("Il programma continua...")
```

Questa effettua una semplice stampa dell'eccezione, per cui il programma non

viene interrotto e dopo la stampa prosegue la sua normale esecuzione:

```
$ python foo.py
Traceback (most recent call last):
File "foo.py", line 4, in <module>
print(mytuple[5]) # Solleva una eccezione
IndexError: tuple index out of range
Il programma continua...
```

Per stampare il traceback, lo si passa alla funzione `traceback.print_tb()`. Come sappiamo, il traceback è il terzo elemento della tupla restituita da `sys.exc_info()`:

```
$ cat foo.py
import traceback
import sys
mytuple = (1, 2, 3)
try:
print(mytuple[5]) # Solleva una eccezione
except:
traceback.print_tb(sys.exc_info()[2]) # Stampa il traceback

print("Il programma continua...")
```

Come possiamo vedere, viene stampato il traceback, dopodiché il programma continua:

```
$ python foo.py
File "foo.py", line 5, in <module>
print(mytuple[5]) # Solleva una eccezione
Il programma continua...
```

Il problema delle etichette che non si riescono ad assegnare nella clausola try

Visto che non possiamo uscire dal ciclo `while` neppure con una interruzione da tastiera, potremmo fare in modo che l'inserimento della stringa `'quit'` porti all'esecuzione di una istruzione `break`, così da poter uscire dal ciclo `while`:

```
>>> import sys
>>> mylist = ['a', 'b', 'c']
>>> while True:
... try:
... i = int(input("Inserisci un numero intero: "))
... print(mylist[i])
... print('Siamo nella suite della try')
... except IndexError:
```



```

... print('Indice non valido!')
... except ValueError:
... if(i == 'quit'): # Errore: viene sollevata una nuova eccezione!
... break
... print('Non riesco a convertire in numero il testo inserito.')
... except:
... print('\nSiamo nella suite della except generica')
... print('Tipo di eccezione:', sys.exc_info()[0])
... print('Istanza:', sys.exc_info()[1])
... print('Siamo al di fuori del blocco try/except')
...
Inserisci un numero intero: quit
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: invalid literal for int() with base 10: 'quit'

```

During handling of the above exception, another exception occurred:

```

Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
NameError: name 'i' is not defined

```

Quando si inserisce 'quit' (o qualsiasi altra stringa che non può essere convertita in intero da `int()`), il programma effettivamente termina, ma non in modo normale come ci aspettavamo, bensì a causa di una nuova eccezione sollevata nella suite della `except` generica. Infatti, quando abbiamo inserito la stringa 'quit', la chiamata `int('quit')` ha sollevato una eccezione di tipo `ValueError` e quindi non è stato assegnato alcun oggetto all'etichetta `i`, che pertanto non è stata definita. Il flusso di esecuzione è passato alla `except` che gestisce le eccezioni di tipo `ValueError`, ma l'istruzione `if(i == 'quit')` ha sollevato una nuova eccezione di tipo `NameError`, visto che l'etichetta `i` non è definita, per cui il programma è terminato. Il problema può essere risolto convertendo la stringa in intero solo dopo che questa è stata assegnata, potendola così utilizzare poi nella `except` per effettuare il confronto:

```

>>> import sys
>>> mylist = ['a', 'b', 'c']
>>> while True:
... try:
... text = input("Inserisci un numero intero: ")
... i = int(text)
... print(mylist[i])

```

```

... print('Siamo nella suite della try')
... except IndexError:
... print('Indice non valido!')
... except ValueError:
... if(text == 'quit'):
... break
... print('Non riesco a convertire in numero il testo inserito.')
... except:
... print('\nSiamo nella suite della except generica')
... print('Tipo di eccezione:', sys.exc_info()[0])
... print('Istanza:', sys.exc_info()[1])
... print('Siamo al di fuori del blocco try/except')
...
Inserisci un numero intero: quit
>>>

```

Dobbiamo, quindi, prestare molta attenzione quando utilizziamo nella suite della `except` etichette definite nella suite della `try`.

La clausola `finally`

La clausola `finally` è molto utile quando ci si vuole assicurare che alcune azioni vengano eseguite, qualunque cosa accada. Nell'esempio seguente non viene sollevata alcuna eccezione e la suite della `finally` viene eseguita:

```

>>> try:
... a = 33
... finally:
... print('Siamo nella suite della finally')
...
Siamo nella suite della finally

```

Qua viene sollevata una eccezione, gestita poi nella `except`, e la suite della `finally` viene comunque eseguita:

```

>>> try:
... int('python')
... except:
... print('Eccezione catturata e gestita!')
... finally:
... print('Siamo nella suite della finally')
...
Eccezione catturata e gestita!

```

Siamo nella suite della finally

Anche se l'eccezione non viene gestita, la `finally` viene comunque eseguita:

```
>>> try:
...     int('python')
... except IndexError:
...     print('Siamo nella except della IndexError')
... finally:
...     print('Siamo nella suite della finally')
...
Siamo nella suite della finally
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'python'
```

Se la `except` cattura l'eccezione ma ne solleva un'altra, la `finally` viene comunque eseguita:

```
>>> try:
...     int('python')
... except:
...     print('Siamo nella except')
...     1 + [1] # Solleva una eccezione di tipo TypeError
...     print('Dopo 1 + [1]')
... finally:
...     print('Siamo nella suite della finally')
...
Siamo nella except
Siamo nella suite della finally
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'python'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Si osservi che quando nella suite di una clausola `except` viene sollevata una nuova eccezione, allora viene mostrato sia il messaggio di errore dell'eccezione catturata, sia quello della nuova eccezione sollevata.

Infine, se siamo nel corpo di una funzione e nella suite della `try` restituiamo il controllo al chiamante, la suite della `finally` viene ugualmente eseguita:

```
>>> def foo():
...     try:
...         print('In foo')
...         return 'Restituisco al chiamante...'
...     finally:
...         print('In finally')
...
>>> foo()
In foo
In finally
'Restituisco al chiamante...'
```

Morale della favola, la suite della clausola `finally` viene sempre e comunque eseguita.

NOTA

La clausola `finally` deve seguire la `except`, se questa è presente:

```
>>> try:
...     int('python')
...     finally:
...         print('Nella finally')
... except:
...     File "<stdin>", line 5
...     except:
...     ^
SyntaxError: invalid syntax
```

Quindi, se ci si vuole accertare che un'azione venga eseguita, come ad esempio la chiusura di un file o di un socket, dobbiamo inserirla nella suite della clausola `finally`. Tra breve vedremo che con l'istruzione `with/as` possiamo ottenere un comportamento analogo.

La clausola opzionale else

La clausola `else` è opzionale e si può inserirla solo dopo le clausole `except`. Se presente, allora la sua suite viene eseguita solo se non viene sollevata alcuna eccezione nella `try`:

```
>>> try:
...     a = 44
... except:
```

```
... print('Nella suite della except...')
... else:
... print("Non è stata sollevata alcuna eccezione nella suite della try")
...
Non è stata sollevata alcuna eccezione nella suite della try
```

Può essere usata solo se è presente una `except` e va sempre posta prima della `finally`:

```
>>> try:
... a = 33
... except:
... print('In except...')
... else:
... print('In else...')
... finally:
... print('In finally...')
...
In else...
In finally...
```

La clausola `else`, quindi, consente di eseguire delle azioni prima della finalizzazione, senza essere costretti a inserirle nella suite della `try`. Vedremo un esempio di utilizzo nella sezione *Esempio pratico sulla gestione delle eccezioni*.

Sollevare le eccezioni manualmente

Le istruzioni `raise` e `assert` ci consentono di sollevare delle eccezioni, la prima di qualsiasi tipo, mentre la seconda solamente di tipo `AssertionError`.

L'istruzione raise

Se vogliamo sollevare una eccezione manualmente, possiamo utilizzare l'istruzione `raise`. Se indichiamo il tipo dell'eccezione, verrà sollevata un'eccezione creata chiamando la classe senza argomenti, per cui l'attributo `args` dell'eccezione sarà una tupla vuota e quindi nel messaggio di errore non comparirà la descrizione della causa:

```
>>> for i in range(10):
... print(i)
... raise IndexError # Solleva IndexError()
...
0
```

```
Traceback (most recent call last):
```

```
...
```

```
IndexError
```

Se, invece, creiamo direttamente una istanza, possiamo passare alla classe la causa dell'errore:

```
>>> for i in range(10):
```

```
...     print(i)
```

```
...     raise IndexError('Esempio di eccezione sollevata manualmente')
```

```
...
```

```
0
```

```
Traceback (most recent call last):
```

```
...
```

```
IndexError: Esempio di eccezione sollevata manualmente
```

L'istruzione `raise`, all'interno della suite di una clausola `except`, può essere utilizzata anche senza indicare un tipo di eccezione o una istanza. In questo caso `raise` solleva l'eccezione catturata (si dice che *propaga* l'eccezione):

```
>>> try:
```

```
...     (1, 2, 3)[5] # L'elemento di indice 5 non esiste...
```

```
... except:
```

```
...     print('Eccezione catturata')
```

```
...     raise
```

```
...
```

```
Eccezione catturata
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in <module>
```

```
IndexError: tuple index out of range
```

L'istruzione `raise` può essere usata anche con la parola chiave `from`, in modo da sollevare una nuova eccezione a partire da quella vecchia:

```
>>> try:
```

```
...     raise IndexError("Causa dell'errore...")
```

```
... except IndexError as ex:
```

```
...     raise TypeError("Giusto per fare un esempio...")
```

```
...
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in <module>
```

```
IndexError: Causa dell'errore...
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
```

File "<stdin>", line 4, in <module>
TypeError: Giusto per fare un esempio...

Come possiamo vedere, le due eccezioni sono legate (*exception chaining*), per cui viene mostrato il messaggio di errore di entrambe.

Questa variante dell'istruzione `raise` è utile quando abbiamo dei vincoli sull'interfaccia. Vediamo un esempio pratico: stiamo lavorando nell'ambito di un progetto che coinvolge più persone e dobbiamo creare una funzione `fromsquared()` che prende due argomenti, una sequenza `seq` e un indice `index`, e restituisce l'elemento `seq[index**2]`. L'indice può essere un numero o una stringa. Scriviamo, quindi, la seguente implementazione di `fromsquared()`:

```
>>> def fromsquared(seq, index):
...     """Restituisci l'elemento della sequenza di indice `index ** 2`"""
...     i = int(index)
...     return seq[i**2]
...
>>> fromsquared(['a', 'b', 'c', 'd', 'e'], 2)
'e'
>>> fromsquared(['a', 'b', 'c', 'd', 'e'], 0)
'a'
>>> fromsquared(['a', 'b', 'c', 'd', 'e'], 1)
'b'
>>> fromsquared(['a', 'b', 'c', 'd', 'e'], 2)
'e'
```

Ci viene detto che i nostri collaboratori, che utilizzeranno la funzione `fromsquared()`, si aspettano che questa, nel caso in cui non sia possibile indicizzare la sequenza, sollevi solo eccezioni di tipo `IndexError`, in modo da poter gestire gli errori con una `except IndexError`:

```
>>> try:
...     fromsquared(['a', 'b', 'c', 'd', 'e'], 3)
... except IndexError as ex:
...     print('Catturata!', ex)
...
Catturata! list index out of range
```

Il problema è che la nostra funzione solleva anche eccezioni di tipo `ValueError`:

```
>>> try:
...     fromsquared(['a', 'b', 'c', 'd', 'e'], 'due')
... except IndexError:
...     print('Catturata!', ex)
```

```
...
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'due'
```

e di tipo `TypeError`:

```
>>> try:
...     fromsquared(['a', 'b', 'c', 'd', 'e'], 1 + 2j)
... except IndexError:
...     print('Catturata!', ex)
...
Traceback (most recent call last):
...
TypeError: can't convert complex to int
```

Possiamo risolvere il problema con `raise from`, catturando le eccezioni di tipo `ValueError` e `TypeError` e sollevando, a partire da queste, delle nuove eccezioni di tipo `IndexError`:

```
>>> def fromsquared(seq, index):
...     """Restituisci l'elemento della sequenza di indice `index ** 2`"""
...     try:
...         i = int(index)
...         return seq[i**2]
...     except ValueError as ex:
...         raise IndexError('La stringa non rappresenta un numero intero') from ex
...     except TypeError as ex:
...         raise IndexError("L'indice deve essere una stringa, un int o un float") from ex
...     except Exception as ex:
...         raise IndexError('Errore non previsto') from ex
...
>>> try:
...     fromsquared(['a', 'b', 'c', 'd', 'e'], 'due')
... except IndexError as ex:
...     print(type(ex), ex)
...
<class 'IndexError'> La stringa non rappresenta un numero intero
>>> try:
...     fromsquared(['a', 'b', 'c', 'd', 'e'], 1 + 2j)
... except IndexError as ex:
...     print(type(ex), ex)
...
<class 'IndexError'> L'indice deve essere una stringa, un int o un float
```


Quando catturiamo una eccezione e nella suite della `except` ne solleviamo una nuova, le due eccezioni vengono sempre legate l'una all'altra. Se non vogliamo tenere traccia dell'eccezione originale, possiamo utilizzare l'istruzione `raise from` indicando `None` come eccezione di partenza:

```
>>> try:
...     raise IndexError("Causa dell'errore...")
... except IndexError:
...     raise TypeError("Giusto per fare un esempio...") from None
...
Traceback (most recent call last):
...
TypeError: Giusto per fare un esempio...
```

Per maggiori informazioni sulle *exception chaining*, possiamo consultare la PEP-0344.

L'istruzione assert

L'istruzione `assert`, che abbiamo visto nel [Capitolo 1](#), solleva una eccezione di tipo `AssertionError` quando la sua espressione di test è valutata come `False`:

```
>>> assert 2 * 2 == 4
>>> assert 2 * 2 == 3
Traceback (most recent call last):
...
AssertionError
```

Dopo l'espressione di test è possibile indicare una ulteriore espressione, che viene passata come argomento ad `AssertionError` quando si crea l'eccezione:

```
>>> try:
...     assert 2 * 2 == 3, "Non mi tornano i conti..."
... except AssertionError as ex:
...     print(ex.args)
...     raise
...
('Non mi tornano i conti...')
Traceback (most recent call last):
...
AssertionError: Non mi tornano i conti...
```

L'utilizzo dell'istruzione `assert` è limitato ai soli casi in cui si vogliono effettuare delle verifiche molto semplici e immediate nel codice, unicamente durante la fase di debug. Per questo motivo, le istruzioni `assert` vengono valutate

solamente quando l'etichetta globale `__debug__` è `True`.
Consideriamo il seguente script:

```
$ cat foo.py
import sys
print("__debug__:", __debug__)
def foo(value):
    assert value not in ['php', 'java']
    return "Ma che bello programmare in {}".format(value)

print(foo(sys.argv[1]))
```

Ecco cosa accade quando lo eseguiamo:

```
$ python foo.py Python3
__debug__: True
Ma che bello programmare in Python3!
```

Come possiamo vedere, per default `__debug__` è `True`, per cui le istruzioni `assert` vengono valutate:

```
$ python foo.py php
__debug__: True
Traceback (most recent call last):
...
AssertionError
```

Lo switch `-O`, come abbiamo visto nel [Capitolo 1](#), genera del bytecode ottimizzato e ha l'effetto di disabilitare la modalità di debug:

```
$ python -O foo.py php # Il secondo output ha lo stesso valore di __debug__...
__debug__: False
Ma che bello programmare in php!
```

Le istruzioni `assert`, anche se molto intuitive, non dovrebbero essere usate per sollevare eccezioni, per due motivi. Il primo è che dobbiamo sempre sollevare l'eccezione del tipo più appropriato e non una poco indicativa `AssertionError`. Il secondo, decisamente più importante del primo, è che, se usiamo delle istruzioni `assert` il programma funziona in modo diverso a seconda che si generi codice ottimizzato o meno.

Consideriamo, ad esempio, il seguente script:

```
$ cat wrongassert.py
assert hasattr(str, 'foo'), "La classe str non ha l'attributo 'foo'"
```

Questo nella situazione di default solleva una `AssertionError`:

```
$ python wrongassert.py
Traceback (most recent call last):
...
AssertionError: La classe str non ha l'attributo `foo`
```

Se, però, vogliamo ottimizzare il nostro codice, allora l'eccezione non viene sollevata:

```
$ python -O wrongassert.py
$
```

Questo è il motivo per cui l'istruzione `assert` è spesso oggetto di discussione e periodicamente qualcuno propone di rimuoverla dal linguaggio. Continua, però, a esistere, perché chi la usa nel modo corretto sostiene che il fatto che i meno esperti potrebbero usarla nel modo scorretto non è un motivo valido per rimuoverla. Quindi teniamo presente che non dobbiamo mai utilizzare l'istruzione `assert` per verificare delle condizioni di funzionamento che non sono unicamente legate al debug.

Eccezioni ed ereditarietà

Supponiamo di voler creare un nuovo tipo di eccezione:

```
>>> class MyException:
...     pass
```

Ecco ciò che accade se proviamo a sollevare un'eccezione di tipo `MyException`:

```
>>> raise MyException()
Traceback (most recent call last):
...
TypeError: exceptions must derive from BaseException
```

Come possiamo osservare, il tipo di eccezione indicato nel messaggio di errore non è `MyException`, ma `TypeError`. Infatti l'istruzione `raise MyException()` non è andata a buon fine, ma ha sollevato essa stessa un'eccezione di tipo `TypeError`. La causa di questa eccezione è che l'istruzione `raise` può sollevare solamente eccezioni il cui tipo deriva da `BaseException` (*exceptions must derive from BaseException*). Il nostro tipo di eccezione, effettivamente, non è stato derivato da `BaseException`.

Vediamo cosa succede se proviamo a catturare una eccezione di tipo `MyException`:

```
>>> try:
...     raise IndexError()
... except MyException:
...     print('Nella suite della except...')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: catching classes that do not inherit from BaseException is not allowed
```

Come possiamo vedere, è stata sollevata una eccezione di tipo `IndexError`. Il flusso di esecuzione è quindi passato alla clausola `except`, la quale ha verificato il tipo dell'eccezione sollevata. Questo test ha però sollevato una seconda eccezione, poiché non è possibile catturare eccezioni il cui tipo non deriva da `BaseException` (*catching classes that do not inherit from BaseException is not allowed*). Quindi, in definitiva, se vogliamo creare un nuovo tipo di eccezione, dobbiamo necessariamente ereditare da `BaseException`.

In realtà, per i motivi che vedremo tra breve, non dobbiamo ereditare direttamente da `BaseException`, ma da una sua sottoclasse, chiamata `Exception`, o da una classe derivata da `Exception`:

```
>>> class MyException(Exception):
...     pass
...
>>> issubclass(MyException, BaseException)
True
```

Il nostro nuovo tipo di eccezione è, quindi, una sottoclasse di `BaseException`, per cui adesso possiamo utilizzarlo sia per sollevare eccezioni sia per catturarle:

```
>>> try:
...     raise MyException()
... except MyException:
...     print(sys.exc_info()[2])
...
(<class '__main__.MyException'>, MyException())
```

NOTA

In Python 2, a differenza di Python 3, è possibile istanziare eccezioni di qualunque tipo, per cui non è necessario ereditare da `BaseException`. Ecco un esempio con Python 2.7:

```
>>> class MyException:
...     pass
...
>>> try:
...     raise MyException()
... except MyException:
...     print('Catturata!')
...
Catturata!
```

Per maggiori informazioni si veda la PEP-0352.

La clausola `except TypeA` non cattura solamente le eccezioni del tipo `TypeA`, ma anche quelle dei tipi da esso derivati:

```
>>> class TypeA(Exception):
...     pass
...
>>> class TypeB(TypeA):
...     pass
...
>>> try:
...     raise TypeB()
... except TypeA:
...     print('Catturata!')
...
Catturata!
>>> class TypeC(TypeB):
...     pass
...
>>> try:
...     raise TypeC()
... except TypeA:
...     print('Catturata!')
...
Catturata!
```

In altre parole, la clausola `except TypeA` cattura le eccezioni di tipo `TypeA` e anche tutte le eccezioni di tipo `TypeX`, con `TypeX` sottoclasse di `TypeA`.

In virtù di quanto abbiamo appena detto, è chiaro che risulta importante conoscere la struttura gerarchica dei tipi di eccezione built-in.

La gerarchia dei tipi di eccezione built-in

I tipi di eccezione built-in sono strutturati gerarchicamente nel seguente modo:

```
BaseException
+----- SystemExit
+----- KeyboardInterrupt
+----- GeneratorExit
+----- Exception
+----- StopIteration
+----- ArithmeticError
| +----- FloatingPointError
| +----- OverflowError
| +----- ZeroDivisionError
+----- AssertionError
+----- AttributeError
+----- BufferError
+----- EOFError
+----- ImportError
+----- LookupError
| +----- IndexError
| +----- KeyError
+----- MemoryError
+----- NameError
| +----- UnboundLocalError
+----- OSError
| +----- BlockingIOError
| +----- ChildProcessError
| +----- ConnectionError
| | +----- BrokenPipeError
| | +----- ConnectionAbortedError
| | +----- ConnectionRefusedError
| | +----- ConnectionResetError
+----- FileExistsError
| +----- FileNotFoundError
| +----- InterruptedError
| +----- IsADirectoryError
| +----- NotADirectoryError
| +----- PermissionError
| +----- ProcessLookupError
| +----- TimeoutError
```

```
+----- ReferenceError
+----- RuntimeError
| +----- NotImplementedError
+----- SyntaxError
| +----- IndentationError
| +----- TabError
+----- SystemError
+----- TypeError
+----- ValueError
| +----- UnicodeError
| +----- UnicodeDecodeError
| +----- UnicodeEncodeError
| +----- UnicodeTranslateError
+----- Warning
+----- DeprecationWarning
+----- PendingDeprecationWarning
+----- RuntimeWarning
+----- SyntaxWarning
+----- UserWarning
+----- FutureWarning
+----- ImportWarning
+----- UnicodeWarning
+----- BytesWarning
+----- ResourceWarning
```

In accordo con quanto detto prima, tutti i tipi di eccezione sono sottoclassi di `BaseException`. Inoltre, possiamo osservare che non tutti i tipi di eccezione terminano con `Error`. Questo perché non tutte le eccezioni rappresentano degli errori. Vediamo, quindi, di chiarire il significato dei vari tipi.

Eccezioni che rappresentano degli errori

Le eccezioni dei tipi che terminano con `Error` vengono sollevate quando si manifestano degli errori. Appartengono a questa categoria, ad esempio, le eccezioni di tipo `NameError`:

```
>>> print(foo) # Non abbiamo definito l'etichetta foo
Traceback (most recent call last):
...
NameError: name 'foo' is not defined
```

quelle di tipo `TypeError`:

```
>>> [1] + 'ciao'
Traceback (most recent call last):
...
TypeError: can only concatenate list (not "str") to list
```

❷ IndexError:

```
>>> mylist = [1, 2, 3]
>>> mylist[5]
Traceback (most recent call last):
...
IndexError: list index out of range
```

Come sappiamo, gli errori, se non vengono gestiti, fanno terminare l'esecuzione del programma.

Eccezioni utilizzate per notificare degli avvisi

I tipi che terminano con `Warning` vengono utilizzati per notificare degli avvisi e tipicamente si limitano a questo. Ad esempio, se eseguiamo una comparazione tra byte e stringhe, non otteniamo alcun errore e non ci viene mostrato alcun avviso:

```
$ python -c 'print(b"python" == "python")'
False
```

In realtà viene generato un avviso che per default non viene notificato. Possiamo abilitare la notifica dei `BytesWarning` utilizzando lo switch `-b`:

```
$ python -b -c 'print(b"python" == "python")'
-c:1: BytesWarning: Comparison between bytes and string
False
```

Possiamo anche fare in modo che i `BytesWarning` siano trattati come degli errori, utilizzando, questa volta, lo switch `-bb`:

```
$ python -bb -c 'print(b"python" == "python")'
Traceback (most recent call last):
...
BytesWarning: Comparison between bytes and string
```

Un particolare tipo di warning, che merita un occhio di riguardo, è costituito dai `DeprecationWarning`, utilizzati per notificare le feature che sono deprecate. A differenza di Python 2, in Python 3 per default i `DeprecationWarning` non vengono visualizzati. Ad esempio, a partire da Python 3.3 la funzione `platform.popen()` è deprecata a favore delle funzioni del modulo `subprocess`, ma se la usiamo non ci viene mostrato alcun messaggio di warning:

```
$ python -c "import platform; platform.popen('ls')"
```


\$

Se, però, utilizziamo lo switch `-w` con l'argomento `all`, allora vengono mostrati tutti i warning:

```
$ python -W all -c "import platform; platform.popen('ls')"  
-c:1: DeprecationWarning: use os.popen instead  
-c:1: ResourceWarning: unclosed file <_io.TextIOWrapper name=3 encoding='UTF-8'>
```

Inoltre, con `-W error` i warning vengono considerati errori:

```
$ python -W error -c "import platform; platform.popen('ls')"  
Traceback (most recent call last):  
...  
DeprecationWarning: use os.popen instead
```

Se vogliamo sollevare dei warning, dobbiamo farlo attraverso la funzione `warnings.warn()`:

```
$ cat foo.py  
import warnings  
warnings.warn("Attenzione, questo è deprecato", DeprecationWarning)
```

```
$ python foo.py  
$ python -W all foo.py  
foo.py:2: DeprecationWarning: Attenzione, questo è deprecato  
  warnings.warn("Attenzione, questo è deprecato", DeprecationWarning)  
$ python -W error foo.py  
Traceback (most recent call last):  
...  
DeprecationWarning: Attenzione, questo è deprecato
```

Per maggiori informazioni sui warning, si consulti la documentazione del modulo `warnings` della libreria standard, alla pagina <http://docs.python.org/3/library/warnings.html>.

NOTA

Se usiamo Python 2.6 o 2.7 e vogliamo vedere i warning relativi a ciò che è deprecato perché cambierà con Python 3, dobbiamo eseguire Python con l'opzione `-3`. Consideriamo, ad esempio, il seguente file:

```
$ cat foo.py  
print 10/3
```

Se lo eseguiamo normalmente, non viene mostrato alcun avviso:

```
$ python2.7 foo.py
3
```

Se usiamo l'opzione `-3`, viene mostrato un warning:

```
$ python2.7 -3 foo.py
foo.py:1: DeprecationWarning: classic int division
print 10/3
3
```

Come al solito, usando `-W error`, il warning viene considerato errore:

```
$ python2.7 -3 -W error foo.py
Traceback (most recent call last):
  File "foo.py", line 1, in <module>
    print 10/3
DeprecationWarning: classic int division
```

Le eccezioni di tipo `StopIteration`

Le eccezioni di tipo `StopIteration`, come abbiamo visto al termine del [Capitolo 1](#), vengono sollevate quando un iteratore ha esaurito gli elementi sui quali iterare:

```
>>> mylist = [1]
>>> i = iter(mylist)
>>> next(i)
1
>>> next(i)
Traceback (most recent call last):
...
StopIteration
```

Sia la classe `StopIteration`, sia le classi che rappresentano degli errori, sia quelle che rappresentano dei warning, ereditano da `Exception` e non direttamente da `BaseException`.

I tipi di eccezione che non ereditano da `Exception`

Le classi `SystemExit`, `KeyboardInterrupt` e `GeneratorExit` sono le uniche a non ereditare direttamente da `BaseException`. Nelle prossime sezioni vedremo il motivo di questa scelta progettuale; per ora ci limitiamo a mostrarne il comportamento. Abbiamo già incontrato le eccezioni di tipo `KeyboardInterrupt`. Queste servono per

interrompere il programma e vengono sollevate quando l'utente preme la combinazione di tasti Control-C.

Le eccezioni di tipo `SystemExit` servono, invece, per uscire dal programma:

```
>>> raise SystemExit()
$
```

Come possiamo vedere, non viene mostrato alcun messaggio di errore.

Le `SystemExit` restituiscono alla shell il codice di errore passato alla classe al momento della creazione dell'istanza:

```
$ python -c "raise SystemExit(0)"; echo $?
0
$ python -c "raise SystemExit(255)"; echo $?
255
$ python -c "import sys; sys.exit(255)"; echo $?
255
```

Se il codice non è un numero intero, allora viene stampato e come codice di errore viene restituito 1:

```
$ python -c "raise SystemExit(3.33)"; echo $?
3.33
1
$ python -c "raise SystemExit('fooooo')"; echo $?
fooooo
1
```

Queste eccezioni vengono sempre sollevate tramite la funzione `sys.exit()`, per cui, quando eseguiamo una istruzione `raise SystemExit(code)`, questa viene tradotta in `sys.exit(code)`.

NOTA

La clausola `finally` viene eseguita anche quando si esce dal programma con una `SystemExit`:

```
$ cat foo.py
try:
    raise SystemExit(100)
finally:
    print('Nella suite della finally...')
```

```
$ python foo.py
Nella suite della finally...
```

Le eccezioni di tipo `GeneratorExit` vengono sollevate quando un generatore, dopo essere stato avviato, viene esplicitamente chiuso con il suo metodo `close()`:

```
>>> def foo():  
...     for i in range(10):  
...     try:  
...     yield i  
... except GeneratorExit:  
...     print('Catturata!')  
...     raise  
...  
>>> gen.close() # Non solleva l'eccezione perché il generatore non è stato ancora avviato  
>>> gen = foo()  
>>> next(gen)  
0  
>>> gen.close()  
Catturata!
```

Come possiamo vedere, l'eccezione non viene propagata al chiamante. Infatti le `GeneratorExit` vengono gestite internamente da Python, il quale le utilizza per comunicare al generatore che è stato chiuso. Poiché non rappresentano degli errori, non dobbiamo preoccuparci di gestirle: è Python che deve farlo. Quindi, se per qualche motivo in un generatore catturiamo una eccezione di questo tipo, dobbiamo ricordarci di propagarla, altrimenti otterremo un errore:

```
>>> def foo():  
...     for i in range(10):  
...     try:  
...     yield i  
... except:  
...     print('Catturata!')  
...     # raise  
>>> gen = foo()  
>>> next(gen)  
0  
>>> gen.close()  
Catturata!  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
RuntimeError: generator ignored GeneratorExit
```

Come catturare tutte le eccezioni in modo generico

Il fatto che la classe `BaseException` sia in cima alla gerarchia delle eccezioni implica che, se nella clausola `except` indichiamo di voler catturare le eccezioni di tipo `BaseException`, otteniamo lo stesso comportamento della clausola `except` generica, ovvero catturiamo ogni tipo di eccezione, comprese le `SystemExit`, le `GenerationExit` e le `KeyboardInterrupt`:

```
>>> import traceback
>>> while True:
...     try:
...         text = input('### ')
...         if text == 'quit':
...             break
...         eval(text)
...     except BaseException:
...         traceback.print_exc()
...
### abcd
Traceback (most recent call last):
  File "<stdin>", line 6, in <module>
  File "<string>", line 1, in <module>
NameError: name 'abcd' is not defined
### Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
KeyboardInterrupt
###
```

In questo caso, possiamo uscire dal ciclo `while` solamente se digitiamo `quit`. Questo tipicamente non è il comportamento desiderato. Infatti abbiamo appena visto che non dobbiamo catturare le `GeneratorExit`, e lo stesso discorso vale per le `KeyboardInterrupt` e `SystemExit`. Queste ultime, infatti, non vanno catturate, visto che vengono intenzionalmente sollevate, rispettivamente, per interrompere il programma e per uscire dal programma.

Per questo motivo, se vogliamo catturare tutte le eccezioni inaspettate, o comunque tutte le eccezioni nel modo più generico possibile, non dobbiamo usare la clausola `except` generica o `except BaseException`, ma limitarci a catturare le sole eccezioni di tipo `Exception`:

```
>>> import traceback
>>> while True:
...     try:
...         text = input('### ')
...         eval(text)
...     except Exception:
```

```

... eval(text)
... except Exception:
...     traceback.print_exc()
...
### abc
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
  File "<string>", line 1, in <module>
NameError: name 'abc' is not defined
### Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
KeyboardInterrupt
>>>

```

Usare il tipo più specializzato

Se vogliamo definire dei tipi di eccezione che non servono per far terminare appositamente il programma, dobbiamo ereditare da `Exception`, o da una sua classe derivata, piuttosto che da `BaseException`. Così facendo, le eccezioni si comportano come ci si aspetta e vengono catturate dalla clausola `except Exception`:

```

>>> class MyException(Exception):
...     pass
...
>>> try:
...     raise MyException()
... except Exception:
...     print('Catturata!')
...
Catturata!

```

Se avessimo ereditato direttamente da `BaseException`, non saremmo riusciti a catturarla:

```

>>> class MyException(BaseException):
...     pass
...
>>> try:
...     raise MyException()
... except Exception:
...     print('Catturata!')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.MyException

```

La regola generale è che dobbiamo sempre ereditare dal tipo più specializzato.

Quindi, se vogliamo definire un tipo di eccezione che rappresenta un errore di indicizzazione, ereditiamo da `IndexError` e non da `Exception`. In questo modo, le clausole `except IndexError` sono in grado di gestire anche la nostra eccezione:

```
>>> class MyIndexError(IndexError):
...     pass
...
>>> try:
...     raise MyIndexError()
... except IndexError:
...     print('Catturata...')
...
Catturata...
```

Questo non sarebbe stato possibile se avessimo ereditato da `Exception`:

```
>>> class MyIndexError(Exception):
...     pass
...
>>> try:
...     raise MyIndexError()
... except IndexError:
...     print('Catturata...')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.MyIndexError
```

L'altra regola generale è quella di catturare le eccezioni del tipo più specializzato. Quindi, se vogliamo gestire eccezioni di tipo `TypeError`, usiamo una clausola `except TypeError` e non `except Exception`, altrimenti potremmo catturare eccezioni inaspettate, come `IndexError`, `NameError` ecc., e ci troveremmo a gestirle come se fossero dei `TypeError`. Secondo la legge di Murphy, questa cattiva gestione sarà causa di errori logici.

Tutte le eccezioni hanno l'attributo args

Nella sezione *La forma except as per avere le informazioni sull'istanza* abbiamo detto che tutte le eccezioni hanno l'attributo `args`, il quale è una tupla contenente il messaggio di errore. Quando definiamo un nuovo tipo di eccezione, le sue istanze hanno anch'esse l'attributo `args`, poiché questo viene ereditato da `BaseException`:

```
>>> hasattr(BaseException, 'args')
```

```

True
>>> class MyException(Exception):
...     pass
...
>>> ex = MyException()
>>> ex.args
()
>>> ex = MyException('Messaggio di errore...')
>>> ex.args
('Messaggio di errore...',)
>>> print(ex)
Messaggio di errore...

```

Gli argomenti che passiamo alla classe al momento della creazione dell'istanza vengono inseriti in `args` anche se facciamo l'overwriting dell'inizializzatore:

```

>>> class MyException(Exception):
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
...
>>> ex = MyException([1, 2, 3], 'python')
>>> ex.args
([1, 2, 3], 'python')
>>> print(ex)
([1, 2, 3], 'python')

```

Esempio pratico sulla gestione delle eccezioni

Consideriamo il seguente esempio di codice:

```

$ cat command_input.py
import subprocess
from sympy import *

namespace = {}
while True:
    raw_line = input(">> ").strip()
    fline = raw_line.format_map(namespace)
    if fline == 'quit':
        break
    elif not fline:
        continue
    elif fline.startswith('!'):
        subprocess.call(fline[1:].split())
    elif '=' in fline:
        n, e = [item.strip() for item in fline.split('=')] # n -> name, e -> expression
        namespace.update({n:e}) if n.isalnum() else print("Nome '%s' non valido!" %n)
    elif fline == raw_line and fline.isalnum():
        print("Prova con {%s} piuttosto che con %s." %(fline, fline))
    else:

```



```
print(simplify(fline) if fline else "")
```

Questo script implementa un semplice esecutore di comandi, del quale vedremo una versione object oriented nell'esercizio conclusivo, al termine del capitolo.

Prima di analizzare il codice, vediamo come si comporta il programma. Come possiamo osservare, quando lo eseguiamo ci viene mostrato un prompt dei comandi:

```
$ python command_input.py
>>
```

Se diamo un comando preceduto da `!`, allora il comando viene cercato nel path di sistema e viene creato un nuovo processo. Ad esempio, `!ls` esegue il comando di sistema `ls`, creando un nuovo processo:

```
>> !ls /home/marco/myvenv/
bin include lib man pyvenv.cfg share
```

Allo stesso modo, `!pwd` esegue il comando `pwd`:

```
>> !pwd
/home/marco/Dropbox/doc/python_book/
```

Se i comandi non sono preceduti da un punto esclamativo, allora vengono considerati espressioni di matematica simbolica. In questo caso lo script prova a semplificare l'espressione:

```
>> 2*x + y + 4*x
6*x + y
```

Le espressioni possono essere anche non banali:

```
>> sqrt(12)
2*sqrt(3)
>> solve(x**2 - 1, x)
[-1, 1]
limit(sin(x)/x, x, 0)
1
```

Possiamo assegnare una espressione a una etichetta, in modo da poterla riutilizzare:

```
>> foo = x + 2*y
>> y + x + {foo}
2*x + 3*y
```

Come possiamo vedere, se vogliamo utilizzare l'espressione a cui `foo` fa riferimento, dobbiamo inserire l'etichetta `foo` tra le parentesi graffe.

Per uscire dal programma usiamo il comando `quit`:

```
>> quit
$
```

Adesso siamo pronti per analizzare il codice. Subito dopo gli `import` iniziali viene creato un dizionario vuoto, assegnato all'etichetta `namespace`, dopodiché si entra nel ciclo `while`. La prima istruzione del ciclo mostra il prompt e assegna all'etichetta `raw_line` la stringa inserita da linea di comando. Supponiamo di inserire la linea `foo = x + 2*y` e vediamo cosa accade. La chiamata a `raw_line.format_map()` in questo caso non ha alcun effetto:

```
>>> namespace = {}
>>> raw_line = 'foo = x + 2*y'
>>> fline = raw_line.format_map(namespace)
>>> fline
'foo = x + 2*y'
```

Quindi `raw_line` è uguale alla `fline` (*formatted line*). Vengono saltate tutte le istruzioni `if` ed `elif` sino ad arrivare a `elif '=' in fline:`. Nella suite di questa clausola `elif`, `fline` viene spezzata in corrispondenza del simbolo di uguale e viene creata la lista `['foo', 'x + 2*y']`:

```
>>> [item.strip() for item in fline.split('=')]
['foo', 'x + 2*y']
```

Il primo elemento della lista viene assegnato a `n` (*name*) e il secondo a `e` (*expression*):

```
>>> n, e = [item.strip() for item in fline.split('=')]
>>> n
'foo'
>>> e
'x + 2*y'
```

A questo punto, visto che il nome è composto da soli caratteri alfanumerici, viene aggiornato il namespace con la coppia `{n:e}`:

```
>>> namespace.update({n:e}) if n.isalnum() else print("Nome '%s' non valido!" %n)
>>> namespace
{'foo': 'x + 2*y'}
```

Il primo ciclo finisce qui e si ritorna al prompt. Supponiamo, ora, di dare il comando `y + x + {foo}`. Il dizionario `namespace` contiene la chiave `'foo'`, per cui questa volta il metodo `str.format_map()` formatta la stringa `raw_line` sostituendo `{foo}` con il corrispondente valore nel `namespace`:

```
>>> raw_line = 'y + x + {foo}'
>>> fline = raw_line.format_map(namespace)
>>> fline
'y + x + x + 2*y'
```

A questo punto il risultato di tutti i test degli `if` ed `elif` è `False`, per cui il flusso di esecuzione entra nel blocco `else`, dove viene stampato il risultato dell'espressione `simplify(fline)`:

```
>>> from sympy import *
>>> print(simplify(fline) if fline else "")
2*x + 3*y
```

La funzione `simplify` è stata importata con l'istruzione `from sympy import *`.

NOTA

SymPy è una libreria di terze parti che si occupa di calcolo simbolico. È scritta interamente in Python e non ha dipendenze, per cui possiamo installarla facilmente. Per maggiori informazioni su SymPy, e più in generale sul calcolo simbolico, possiamo consultare il tutorial online, alla pagina: <http://docs.sympy.org/latest/tutorial/index.html>.

La parte restante del codice non dovrebbe necessitare di chiarimenti. Osserviamo solamente che la funzione `subprocess.call()`, della quale abbiamo già parlato nel [Capitolo 3](#), viene chiamata quando la linea formattata `fline` inizia con `!`. In questo caso `subprocess.call()` esegue il comando creando un nuovo processo e restituendo lo stato di uscita di questo:

```
>>> import subprocess
>>> subprocess.call(['ls', '/home/marco/myvenv'])
bin include lib man pyvenv.cfg share
0
```

Torniamo a noi, ovvero alla gestione delle eccezioni. Tutto, infatti, sembra funzionare a dovere, ma così non è. Proviamo ad avviare il programma e a dare il comando `!cd`:

```
>> !cd
Traceback (most recent call last):
...
FileNotFoundError: [Errno 2] No such file or directory: 'cd'
```

La chiamata a `subprocess.call()` ha sollevato una eccezione perché nel *path* di sistema non è stato trovato alcun file chiamato *cd*. Infatti *cd* è un comando *bash*, non un programma di sistema. I programmi di sistema, come ad esempio *ls*, sono, infatti, dei file eseguibili:

```
$ which cd
$ which ls
/bin/ls
```

Questo significa che, se diamo un comando `!system_command`, ma nel nostro *path* di sistema non esiste alcun programma chiamato *system_command*, allora `subprocess.call()` solleva una eccezione di tipo `FileNotFoundError` e il programma termina la sua esecuzione. Noi vorremmo che il programma non terminasse l'esecuzione, ma stampasse un messaggio di errore e poi mostrasse nuovamente il prompt. Per fare ciò dobbiamo gestire l'eccezione. Nel nostro caso, `subprocess.call()` solleva una eccezione di tipo `FileNotFoundError`, per cui questo è il tipo di eccezione che dobbiamo gestire:

```
elif fline.startswith('!'):
    try:
        subprocess.call(fline[1:].split())
    except FileNotFoundError:
        print('Nome del programma non valido!')
```

Adesso l'errore viene gestito e il programma non termina la sua esecuzione:

```
>> !cd
Nome del programma non valido!
```

Ora proviamo a eseguire il programma *tail*, il quale mostra sullo standard output le ultime linee di dati provenienti da uno o più file di testo o dallo standard input. Visualizziamo, ad esempio, le ultime tre linee (`-n 3`) del file *command_input.py*:

```
>> !tail -n 3 command_input.py
print("Prova con {%s} piuttosto che con %s." %(fline, fline))
else:
print(simplify(fline) if fline else "")
>>
```

Ancora una volta tutto sembra andare bene. Se, però, diamo il comando `tail` senza argomenti, non si ritorna al prompt perché il programma `tail` resta in attesa dell'input:

```
>> !tail
█
```

A partire da Python 3.3 la funzione `subprocess.call()` accetta un argomento opzionale, chiamato `timeout`, utilizzato per indicare un tempo massimo di esecuzione del comando. Se usiamo un `timeout` di `T` secondi e il processo, trascorso questo tempo dalla sua creazione, non è ancora terminato, allora `subprocess.call()` lo uccide e solleva una eccezione di tipo `subprocess.TimeoutExpired`:

```
>>> subprocess.call('tail', timeout=5)
Traceback (most recent call last):
...
subprocess.TimeoutExpired: Command 'tail' timed out after 3 seconds
```

Effettueremo, quindi, la chiamata `subprocess.call(fline[1:].split(), timeout=5)`, preoccupandoci di gestire l'eccezione `subprocess.TimeoutExpired`:

```
elif fline.startswith('!'):
try:
subprocess.call(fline[1:].split(), timeout=5)
except FileNotFoundError:
print('Nome del programma non valido!')
except subprocess.TimeoutExpired:
print('Il processo non è terminato nel timeout stabilito')
```

Ora ciascuna eccezione viene gestita in modo diverso dalle altre:

```
>> !cd
Nome del programma non valido!
>> !tail
Il processo non è terminato nel timeout stabilito
```

Non abbiamo risolto tutto, vi sono altri casi in cui vengono sollevate delle eccezioni. Ad esempio, non è possibile effettuare due assegnamenti nello stesso comando:

```
>> a = b = 33
Traceback (most recent call last):
...
ValueError: too many values to unpack (expected 2)
```

per cui scriviamo:

```
try:
n, e = [item.strip() for item in fline.split('=')] # n -> name, e -> expression
except ValueError:
print("Non è possibile fare più di un assegnamento nello stesso comando")
else:
namespace.update({n:e}) if n.isalnum() else print("Nome '%s' non valido!" %n)
```

Non è possibile neppure tentare di utilizzare una etichetta che non è stata definita:

```
>> x + {myx}
Traceback (most recent call last):
...
KeyError: 'myx'
```

In quest'ultimo caso, visto che non sappiamo quale sia tale l'etichetta, e visto che il nome di questa, quando viene sollevata una eccezione di tipo `KeyError`, è proprio la causa dell'errore, optiamo per una gestione con `except as:`

```
try:
fline = raw_line.format_map(namespace)
except KeyError as ex:
print("L'etichetta %s non è definita" %ex)
```

Infine, tra le parentesi graffe non possiamo inserire un numero intero o floating point:

```
>> {11.3}
Traceback (most recent call last):
...
ValueError: Format string contains positional fields
```

Aggiungiamo, quindi, anche la gestione delle eccezioni di tipo `ValueError`:

```
try:
fline = raw_line.format_map(namespace)
except KeyError as ex:
print("L'etichetta %s non è definita" %ex)
except ValueError:
print("L'etichetta tra parentesi graffe non può essere un numero")
```

Adesso sembra che tutto vada bene:

```
>> a = b = x + 33
Non è possibile fare più di un assegnamento nello stesso comando
>> a = x + 33
>> {a}
x + 33
>> {a} + {b}
L'etichetta 'b' non è definita
>> {44}
L'etichetta tra parentesi graffe non può essere un numero
```

In realtà c'è un intero mondo a noi oscuro, popolato da tutte le eccezioni che può lanciare `simplify()`:

```
>> a = 2(x + y)
>> {a}
Traceback (most recent call last):
...
TypeError: 'Integer' object is not callable
```

In questo caso, ad esempio, eravamo intenzionati a scrivere $a = 2 \cdot (x + y)$, ma ci siamo dimenticati dell'operatore di moltiplicazione. Visto che conosciamo poco SymPy, e quindi non sappiamo quali eccezioni può sollevare `simplify()`, scegliamo di utilizzare la clausola `except Exception` in modo da intercettare tutte le eccezioni non previste e lasciar passare le `SystemExit` e `KeyboardInterrupt`. Ecco, quindi, la versione definitiva del nostro programma:

```
import subprocess
import sys
from sympy import *
namespace = {}
while True:
    try:
        raw_line = input(">> ").strip()
        try:
            fline = raw_line.format_map(namespace)
        except KeyError as ex:
            print("L'etichetta %s non è definita" % ex)
        except ValueError:
            print("L'etichetta tra parentesi graffe non può essere un numero")
        continue
    if fline == 'quit':
        break
    elif not fline:
        continue
    elif fline.startswith('!'):
        try:
            subprocess.call(fline[1:].split(), timeout=5)
        except FileNotFoundError:
```

```

print('Nome del programma non valido!')
except subprocess.TimeoutExpired:
print('Il processo non è terminato nel timeout stabilito')
elif '=' in fline:
try:
n, e = [item.strip() for item in fline.split('=')] # n -> name, e -> expression
except ValueError:
print("Non e' possibile fare piu' di un assegnamento nello stesso comando")
else:
namespace.update({n:e}) if n.isalnum() else \
print("Nome '%s' non valido!" %n)
elif fline == raw_line and fline.isalnum():
print("Prova con {%s} piuttosto che con %s." %(fline, fline))
else:
print(simplify(fline) if fline else "")
except Exception as ex:
print(ex)

```

Adesso riusciamo a gestire nel modo corretto anche le eccezioni non previste:

```

>> 2(x + 3)
'Integer' object is not callable
>> solve(x - cos(x), x)
multiple generators [x, cos(x)]
No algorithms are implemented to solve equation x - cos(x)

```


L'istruzione `with` e i context manager

In questa sezione parleremo dell'istruzione `with` e del concetto di *context manager*.

L'istruzione `with`

Consideriamo una classe `Foo` che definisce due metodi, chiamati `Foo.__enter__()` e `Foo.__exit__()`:

```
>>> class Foo:
...     def __enter__(self):
...         print('Sono %d, sto entrando...' %id(self))
...     def __exit__(self, type, instance, tb):
...         print('Sono %d, sto uscendo...' %id(self))
```

Cerchiamo ora di capire cosa accade quando eseguiamo la seguente istruzione `with`:

```
>>> with Foo():
...     print('Sono nella suite della clausola with')
...
Sono 140672344752656, sto entrando...
Sono nella suite della clausola with
Sono 140672344752656, sto uscendo...
```

Se osserviamo l'output, notiamo che questa istruzione ha causato la chiamata, nell'ordine, di `Foo.__enter__()`, della `print()` interna alla suite e infine di `Foo.__exit__()`. Visto che non siamo stati noi a chiamare `Foo.__enter__()` e `Foo.__exit__()`, deduciamo che è stata l'istruzione `with` a farlo per noi. Infatti il metodo `Foo.__enter__()` è stato automaticamente chiamato prima che si entrasse nella suite e il metodo `Foo.__exit__()` è stato automaticamente chiamato al termine dell'esecuzione della suite.

In altre parole, quando eseguiamo una istruzione `with obj:`

- prima che si entri nella suite viene chiamato il metodo `obj.__enter__()`;
- viene eseguita la suite;
- all'uscita dalla suite viene chiamato il metodo `obj.__exit__()`.

È necessario, quindi, che l'oggetto che viene utilizzato con l'istruzione `with` abbia tali metodi. Nella prossima sezione vedremo in dettaglio come vanno definiti. Il metodo `obj.__exit__()` viene sempre chiamato, anche se nella suite viene sollevata una eccezione:

```
>>> with Foo():
... print('Sono nella suite della clausola with')
... raise Exception('Attenzione, eccezione in arrivo!')
...
Sono 140672344752720, sto entrando...
Sono nella suite della clausola with
Sono 140672344752720, sto uscendo...
Traceback (most recent call last):
...
Exception: Attenzione, eccezione in arrivo!
```

Il precedente codice è quindi equivalente al seguente:

```
>>> f = Foo()
>>> f.__enter__()
Sono 140672344714064, sto entrando...
>>> try:
... print('Sono nella suite della clausola with')
... raise Exception('Attenzione, eccezione in arrivo!')
... finally:
... import sys
... f.__exit__(*sys.exc_info())
...
Sono nella suite della clausola with
Sono 140672344714064, sto uscendo...
Traceback (most recent call last):
...
Exception: Attenzione, eccezione in arrivo!
```

Le istruzioni `with` e `try/finally` sono, quindi, parenti strette, visto che il metodo `__exit__()` esegue la funzione di finalizzazione, allo stesso modo della clausola `finally`.

La clausola as

Osserviamo come nell'esempio introduttivo non sia possibile utilizzare l'istanza:

```
>>> with Foo():
... print('Sono nella suite della clausola with')
...
```

```
Sono 140672344752784, sto entrando...
Sono nella suite della clausola with
Sono 140672344752784, sto uscendo...
```

Se volessimo utilizzare l'istanza, ad esempio per stampare il suo identificativo, potremmo assegnarla a una etichetta prima dell'istruzione `with`:

```
>>> f = Foo()
>>> with f:
... print("Identificativo dell'istanza: %d" %id(f))
...
Sono 140672344752848, sto entrando...
Identificativo dell'istanza: 140672344752848
Sono 140672344752848, sto uscendo...
```

Possiamo ottenere lo stesso risultato in modo più conciso, con la clausola `as`. Per poter fare ciò è necessario che il metodo `Foo.__enter__()` restituisca l'istanza creata:

```
>>> class Foo:
... def __enter__(self):
... print('Sono %d, sto entrando...' %id(self))
... return self
... def __exit__(self, type, instance, tb):
... print('Sono %d, sto uscendo...' %id(self))
```

Come possiamo vedere, l'istanza viene assegnata all'etichetta `f`:

```
>>> with Foo() as f:
... print("Identificativo dell'istanza: ", id(f))
...
Sono 140159919007696, sto entrando...
Identificativo dell'istanza: 140159919007696
Sono 140159919007696, sto uscendo...
```

Infatti la clausola `with Foo() as f` effettua in modo automatico il seguente assegnamento:

```
>>> f = Foo().__enter__()
Sto entrando...
```

Quindi, quando eseguiamo una istruzione `with obj as f`:

- viene chiamato il metodo `obj.__enter__()` e assegnato il risultato all'etichetta `f`;
- viene eseguita la suite;

- all'uscita dalla suite viene chiamato il metodo `obj.__enter__()`.

A partire da Python 3.1, è possibile specificare più clausole `as`:

```
>>> with Foo() as f1, Foo() as f2, Foo() as f3:
...     print("Identificativo dell'istanza: ", id(f1))
...     print("Identificativo dell'istanza: ", id(f2))
...     print("Identificativo dell'istanza: ", id(f3))
...
Sono 140159919008208, sto entrando...
Sono 140159919008464, sto entrando...
Sono 140159919008528, sto entrando...
Identificativo dell'istanza: 140159919008208
Identificativo dell'istanza: 140159919008464
Identificativo dell'istanza: 140159919008528
Sono 140159919008528, sto uscendo...
Sono 140159919008464, sto uscendo...
Sono 140159919008208, sto uscendo...
```

Esempi di utilizzo dell'istruzione `with`

Nel [Capitolo 3](#) abbiamo accennato all'utilizzo dell'istruzione `with` con i file. Infatti i file object sono degli oggetti che hanno i metodi `__enter__()` e `__exit__()`:

```
>>> f = open('myfile')
>>> hasattr(f, '__enter__')
True
>>> hasattr(f, '__exit__')
True
```

Il metodo `f.__exit__()` chiude il file, per cui, se utilizziamo l'istruzione `with`, possiamo essere certi che al termine della suite il file verrà chiuso:

```
>>> with open('myfile') as f:
...     f.readlines()
...
['prima linea\n', 'seconda linea\n']
>>> f.closed
True
```

Il codice appena visto è equivalente al seguente:

```
>>> f = open('myfile').__enter__()
>>> try:
...     f.readlines()
... finally:
...     f.close()
```

```
...
['prima linea\n', 'seconda linea\n']
>>> f.closed
True
```

Se nella suite viene sollevata una eccezione, il file viene ugualmente chiuso:

```
>>> with open('myfile') as f:
...     raise Exception('mmmmm...')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: mmmm...
>>> f.closed
True
```

L'istruzione `with` è quindi particolarmente utile in tutti i contesti in cui è necessario compiere delle operazioni di finalizzazione. Oltre al caso dei file, pensiamo, ad esempio, ai *lock*. Consideriamo l'esempio seguente:

```
>>> import threading
>>> lock = threading.Lock()
>>> def foo():
...     print('In foo')
...     lock.acquire()
...     print('Acquisito!')
...     lock.release()
...     print('Rilasciato!')
...
>>> foo()
In foo
Acquisito!
Rilasciato!
```

Il lock è stato rilasciato, per cui possiamo eseguire nuovamente `foo()`:

```
>>> foo()
In foo
Acquisito!
Rilasciato!
```

Se, però, acquisiamo il lock, e poi, prima di rilasciarlo, si verifica un errore, andiamo in deadlock:

```
>>> def foo():
...     print('In foo')
```

```

... lock.acquire()
... print('Acquisito!')
... raise Exception("Questo sì che è un grosso problema...")
... lock.release()
... print('Rilasciato!')
...
>>> try:
...     foo()
... except Exception as ex:
...     print(ex)
...
In foo
Acquisito!
Questo sì che è un grosso problema...
>>> foo()
In foo

```

I lock hanno i metodi `lock.__enter__()` e `lock.__exit__()`:

```

>>> hasattr(lock, '__enter__')
True
>>> hasattr(lock, '__exit__')
True

```

Il metodo `lock.__exit__()` rilascia il lock, per cui, se scriviamo l'esempio precedente avendo l'accortezza di acquisire il lock con una istruzione `with`, il deadlock non si verificherà più:

```

>>> lock.release() # Se non lo abbiamo ancora fatto, rilasciamo il lock
>>> def foo():
...     print('In foo')
...     with lock: # Chiama automaticamente il metodo acquire()
...         raise Exception("Non è più un grosso problema...")
...     # Al termine della with viene chiamato lock.release()
...
>>> foo()
In foo
Traceback (most recent call last):
...
Exception: Non è più un grosso problema...
>>>

```

Nella prossima sezione vedremo come vanno definiti i metodi `__enter__()` e `__exit__()` affinché un oggetto possa essere utilizzato, nelle modalità che abbiamo

appena visto, in una istruzione `with`.

I context manager

Un *context manager* è un oggetto che ha i metodi `__enter__()` e `__exit__()` definiti in modo tale da poter essere chiamati da una istruzione `with`. La suite dell'istruzione `with` viene anche detta *contesto* della `with`. I context manager possono essere o istanze di una classe oppure possono essere funzioni.

Definire classi che istanziano context manager

Se una classe `Foo` definisce i metodi `Foo.__enter__()` e `Foo.__exit__()`, aventi la seguente signature:

```
>>> class Foo:
...     def __enter__(self):
...         print('In __enter__()')
...     def __exit__(self, type, instance, tb):
...         print('In exit__()')
```

allora le sue istanze sono dei context manager:

```
>>> with Foo():
...     pass
...
In __enter__()
In exit__()
```

Come sappiamo, se utilizziamo una clausola `as`, allora all'etichetta viene assegnato l'oggetto restituito da `Foo.__enter__()`, che nel nostro caso è `None`:

```
>>> with Foo() as f:
...     print(f is None)
...
In __enter__()
True
In exit__()
```

Quindi, come abbiamo già detto, se vogliamo che all'etichetta `f` venga assegnata l'istanza, dobbiamo fare in modo che questa venga restituita da `Foo.__enter__()`:

```
>>> class Foo:
```

```

... def __enter__(self):
...     print('In __enter__(). ID: %d' %id(self))
...     return self
... def __exit__(self, type, instance, tb):
...     print('In exit__(). ID: %d' %id(self))
...
>>> with Foo() as f:
...     print(id(f))
...
In __enter__(). ID: 140115458937104
140115458937104
In exit__(). ID: 140115458937104

```

Al metodo `Foo.__exit__()` vengono sempre passati gli elementi della tupla restituita da `sys.exc_info()`, i quali sono tutti e tre `None` se nel contesto della `with` non viene sollevata alcuna eccezione:

```

>>> class Foo:
...     def __enter__(self):
...         print('In __enter__()')
...         return self
...     def __exit__(self, type, instance, tb):
...         print('In exit__()')
...         print('type:', type)
...         print('instance:', instance)
...         print('tb:', tb, end='\n\n')
...
>>> with Foo():
...     pass
...
In __enter__()
In exit__()
type: None
instance: None
tb: None

```

altrimenti sono, rispettivamente, il tipo di eccezione, l'eccezione e il traceback:

```

>>> with Foo():
...     raise Exception('Attenzione!')
...
In __enter__()
In exit__()
type: <class 'Exception'>
instance: Attenzione!

```



```
tb: <traceback object at 0x7f6f2c284ab8>
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 2, in <module>
```

```
Exception: Attenzione!
```

NOTA

Visto che il metodo `__exit__()` viene chiamato passandogli gli elementi della tupla restituita da `sys.exc_info()`, possiamo definirlo in modo più conciso con un *varargs*:

```
>>> class Foo:
...     def __enter__(self):
...         print('In __enter__()')
...         return self
...     def __exit__(self, *exc):
...         print('In exit__()')
...
>>> with Foo():
...     pass
...
In __enter__()
In exit__()
```

Se il metodo `Foo.__exit__()` restituisce un oggetto che è valutato `True`, allora l'eccezione sollevata nel contesto della `with` non si propaga:

```
>>> class Foo:
...     def __enter__(self):
...         print('In __enter__()')
...         return self
...     def __exit__(self, type, instance, tb):
...         print('In exit__()')
...         print('type:', type)
...         print('instance:', instance)
...         print('tb:', tb)
...         return 100
...
>>> with Foo():
...     raise Exception('Attenzione!')
...
In __enter__()
In exit__()
```

```
type: <class 'Exception'>
instance: Attenzione!
tb: <traceback object at 0x7f6f2c284b00>
```

Come abbiamo detto, i context manager possono essere anche funzioni.

Funzioni context manager

Possiamo creare un context manager decorando una funzione generatore con `contextlib.contextmanager`:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def foo():
...     print('Prima di yield')
...     yield
...     print('Dopo yield')
```

Il metodo `__enter__()` esegue il codice che precede l'istruzione `yield`:

```
>>> f = foo()
>>> f.__enter__()
Prima di yield
```

Il metodo `__exit__()` esegue il codice che segue l'istruzione `yield`:

```
>>> f.__exit__(None, None, None)
Dopo yield
```

Ecco, infatti, cosa accade quando utilizziamo il context manager in una istruzione `with`:

```
>>> with foo():
...     print('Nel contesto della with')
...
Prima di yield
Nel contesto della with
Dopo yield
```

Se, però, viene sollevata una eccezione nel contesto della `with`, il metodo `__exit__()` non viene eseguito:

```
>>> with foo():
...     print('Nel contesto della with')
```

```
... raise Exception('Attenzione, eccezione in arrivo!')
...
Prima di yield
Nel contesto della with
Traceback (most recent call last):
...
Exception: Attenzione, eccezione in arrivo!
```

Se vogliamo che il metodo `__exit__()` venga sempre eseguito, lo inseriamo in una clausola `finally`:

```
>>> @contextmanager
... def foo():
...     try:
...         # Inizia il metodo __enter__()
...         print('Prima di yield')
...         # Finisce il metodo __enter__()
...         yield
...     finally:
...         # Inizia il metodo __exit__()
...         print('Dopo yield')
...
>>> with foo():
...     print('Nel contesto della with')
...     raise Exception('Attenzione, eccezione in arrivo!')
...
Prima di yield
Nel contesto della with
Dopo yield
Traceback (most recent call last):
...
Exception: Attenzione, eccezione in arrivo!
```

Se l'istruzione `yield` produce un oggetto, allora questo, quando si utilizza la clausola `as`, viene assegnato all'etichetta:

```
>>> @contextmanager
... def foo():
...     try:
...         print('Prima di yield')
...         yield 'python'
...     finally:
...         print('Dopo yield')
...
>>> with foo() as f:
```

```
... print(f)
...
Prima di yield
python
Dopo yield
```

La funzione generatore deve produrre un solo oggetto.

NOTA

A partire da Python 3.2, è possibile utilizzare la classe `contextlib.ContextDecorator` per poter definire delle azioni da compiere, sia prima che venga eseguito il corpo di una funzione, sia al termine dell'esecuzione:

```
>>> from contextlib import ContextDecorator
>>> class mycontextdecorator(ContextDecorator):
...     def __enter__(self):
...         print('In __enter__()')
...         return self
...     def __exit__(self, type, instance, tb):
...         print('In __exit__()')
...
>>>
>>> @mycontextdecorator()
... def foo():
...     print('In foo()')
...
>>> foo()
In __enter__()
In foo()
In __exit__()
```

Il metodo `__exit__()` viene sempre chiamato, anche in presenza di una eccezione nel corpo di `foo()`:

```
>>> @mycontextdecorator()
... def foo():
...     print('In foo()')
...     raise Exception('Attenzione!')
...
>>> foo()
In __enter__()
In foo()
In __exit__()
Traceback (most recent call last):
...
```

Per maggiori informazioni possiamo consultare la documentazione ufficiale del modulo `contextlib`, alla pagina <http://docs.python.org/3/library/contextlib.html>.

Esercizio conclusivo

In questo esercizio conclusivo esamineremo una versione *object oriented* del programma che abbiamo incontrato nella sezione *Esempio pratico sulla gestione delle eccezioni*. Il nostro programma è, quindi, un interprete di comandi:

```
$ cat pyWayShell.py
import cmd, subprocess, sys, logging
from sympy import *

logging.basicConfig(filename='pyWayShell.log', level=logging.DEBUG)

class PyWayShell(cmd.Cmd):
    intro = "Benvenuto :) Digita help o ? per vedere l'elenco dei comandi.\n"
    prompt = '>>> '

    f __init__(self):
        cmd.Cmd.__init__(self)
        self.namespace = {}

    f do_quit(self, line):
        """Esce dal programma."""
        print("\nGrazie per aver usato pyWayShell :)")
        sys.exit(1)

    f do_shell(self, line):
        """Esegue il comando nella shell di sistema."""
        try:
            subprocess.call(line, timeout=5, shell=True)
        except subprocess.TimeoutExpired:
            print('Il processo non è terminato nel timeout stabilito')

    f emptyline(self):
        """Non fare nulla se la linea è vuota"""
        pass

    f do_EOF(self, line):
        """Con CTR-D esce dal programma."""
        self.do_quit(line)

    do_q = do_quit

    f precmd(self, line):
        """Azioni prima di interpretare il comando"""
```

```

self.raw_line = line
try:
return line.format_map(self.namespace)
except KeyError as ex:
print("L'etichetta %s non è definita" %ex)
except ValueError:
print("L'etichetta tra parentesi graffe non può essere un numero")

return " # Restituisce una linea vuota

f default(self, line):
"""I comandi non previsti nell'help vengono semplificati con SymPy."""
try:
if '=' in line:
try:
n, e = [item.strip() for item in line.split('=')]
except ValueError:
print("Non è possibile fare più di un assegnamento nello stesso comando")
else:
if n.isalnum():
self.namespace.update({n:e})
else:
print("Nome '%s' non valido!" %n)
elif line == self.raw_line and line.isalnum():
print("Prova con {%s} piuttosto che con %s." %(line, line))
else:
print(simplify(line) if line else "")
except Exception as ex:
logging.debug('Linea: ', line)
logging.debug('Messaggio: %s\n', ex)
print(ex)

f help_default(self):
print(self.default.__doc__)

if __name__ == '__main__':
pws = PyWayShell()
pws.cmdloop()

```

Come al solito, analizzeremo il tutto nelle sezioni che seguono, ma per il momento leggiamo ancora il programma, sforzandoci di capirne il significato in modo autonomo.

Esecuzione del programma

Quando eseguiamo il programma, ci viene mostrato un messaggio di benvenuto, seguito da un prompt dei comandi:

```

$ python pyWayShell.py
Benvenuto :) Digita help o ? per vedere l'elenco dei comandi.
>>

```

Se diamo il comando `help` o `?`, ci viene mostrato l'help:

```
>> help
Documented commands (type help <topic>):
```

```
EOF help q quit shell
```

```
Miscellaneous help topics:
```

```
default
```

I comandi previsti sono: `help` (lo abbiamo appena eseguito), `q`, `quit` e `shell`. Il comando `quit` serve per uscire dal programma:

```
>> help quit
Esce dal programma.
```

Il comando `q` è una scorciatoia di `quit`:

```
>> ? q
Esce dal programma.
```

Il comando `shell` esegue i comandi nella shell di sistema:

```
>> shell head ~/.pyrc
import rlcompleter
import readline
readline.parse_and_bind('tab: complete')
```

Il punto esclamativo è sinonimo di `shell`:

```
>> ! head ~/.pyrc
import rlcompleter
import readline
readline.parse_and_bind('tab: complete')
```

Infine, un comando che non fa parte dei precedenti viene semplificato o risolto con SymPy:

```
>> 2 + 2
4
>> a = x + 44
>> {a} + 2*x -40
3*x + 4
```

Se un comando causa degli errori non previsti, ne viene tenuta traccia

salvando i messaggi di errore in un file di log:

```
>> 2(x + 3)
'Integer' object is not callable
>> solve(x - cos(x), x)
multiple generators [x, cos(x)]
No algorithms are implemented to solve equation x - cos(x)
>> ! head pyWayShell.log
DEBUG:root:Linea: 2(x + 3)
DEBUG:root:Messaggio: 'Integer' object is not callable

DEBUG:root:Linea: solve(x - cos(x), x)
DEBUG:root:Messaggio: multiple generators [x, cos(x)]
No algorithms are implemented to solve equation x - cos(x)
```

Infine, la nostra shell supporta il completamento automatico dei comandi. Ad esempio, se digitiamo il carattere `q` e poi un **TAB**, vengono mostrati i comandi che iniziano con `q`:

```
>> q
q quit
```

Se dopo la `q` digitiamo anche il carattere `u` e poi diamo nuovamente un **TAB**, il comando viene completato con `quit`:

```
>> quit
```

Inoltre, la shell tiene traccia della storia dei comandi (*history*). Ad esempio, con **Control-P** (*previous*, ovvero precedente) si va al comando precedente e con **Control-N** (*next*, successivo) a quello successivo.

Il modulo `cmd` della libreria standard

Il modulo `cmd` della libreria standard fornisce gli strumenti per agevolare la scrittura di programmi che eseguono istruzioni impartite da linea di comando.

La classe `cmd.Cmd`

Il nostro programma consiste in una classe chiamata `PyWayShell`, che eredita dalla classe `cmd.Cmd`. In fondo al file `pyWayShell.py` vi sono le seguenti istruzioni:

```
$ tail -n 3 pyWayShell.py
if __name__ == '__main__':
    pws = PyWayShell()
    pws.cmdloop()
```

L'esecuzione del programma inizia da qui, con la creazione di una istanza di `PyWayShell` e l'esecuzione del metodo `PyWayShell.cmdloop()`. Il metodo `cmdloop()` non è definito in `PyWayShell`, ma viene ereditato dalla classe `cmd.Cmd`:

```
>>> import cmd
>>> hasattr(cmd.Cmd, 'cmdloop')
True
```

Questo metodo, sostanzialmente, stampa il messaggio che abbiamo indicato nell'attributo di classe `PyWayShell.intro`:

```
>>> from pyWayShell import PyWayShell
>>> pws = PyWayShell()
>>> pws.intro
"Benvenuto :) Digita help o ? per vedere l'elenco dei comandi.\n"
```

Fatto ciò, sostanzialmente esegue in loop le seguenti azioni:

- mostra il prompt e resta in attesa che l'utente inserisca un comando;
- passa il comando al metodo `pws.precmd()`, il quale fa eventualmente alcune trasformazioni e poi lo restituisce;
- se `pws.precmd()` ha restituito una stringa vuota, allora chiama il metodo `pws.emptyline()`, il quale, nel nostro caso, non fa nulla. Se `pws.precmd()` inizia con un punto esclamativo, allora chiama `pws.do_shell()` e le passa il comando, altrimenti, se l'utente dà un comando chiamato `foo`, chiama il metodo `pws.do_foo()`, passandogli il comando;
- se il metodo `pws.do_foo()` non esiste, allora chiama il metodo `pws.default()`, passandogli il comando;
- ritorna al primo punto.

Vediamo questi punti uno per uno.

Il prompt è ciò che abbiamo indicato nell'attributo di classe `pws.prompt`:

```
>>> pws.prompt
'>>> '
```

Il metodo `PyWayShell.precmd()` restituisce la linea formatta, che è uguale alla linea inserita dall'utente se non è presente alcuna etichetta tra parentesi graffe:

```
>>> namespace = {'a': '33'}
>>> 'x + y'.format_map(namespace)
'x + y'
>>> 'x + {a}'.format_map(namespace)
'x + 33'
```

Se `str.format_map()` solleva una eccezione, allora `PyWayShell.precmd()` mostra un messaggio e restituisce una stringa vuota. In questo caso `Cmd.cmdloop()` chiama `pws.emptyline()`, il quale non fa nulla.

Se la stringa restituita da `pws.precmd()` non è vuota, allora `Cmd.cmdloop()` ne fa il parsing. Come abbiamo detto, se la linea inizia con un punto esclamativo, allora chiama il metodo `PyWayShell.do_shell()`. Se, invece, non inizia con il punto esclamativo, allora separa gli elementi della linea e considera il primo come il nome del comando. Per intenderci, se la linea è `ls -la`, il comando è `ls`:

```
>>> line = 'ls -la'
>>> cmd, *args = line.split()
>>> cmd
'ls'
>>> args
['-la']
```

A questo punto cerca l'attributo di `pws` che si chiama `'do_%s' %cmd`:

```
>>> 'do_%s' %cmd
'do_ls'
```

Se lo trova, effettua la chiamata `pws.do_ls(line)`, altrimenti effettua la chiamata `pws.default(line)`. La chiamata del metodo della classe figlia, da parte di `Cmd.cmdloop()`, avviene quindi in modo analogo al seguente:

```
>>> class Base:
...     def command(self, line):
...         cmd, *args = [item.strip() for item in line.split()]
...         try:
...             method = getattr(self, 'do_%s' %cmd)
...             method(line)
...         except AttributeError:
...             self.default(line)
...
>>> class Foo(Base):
...     def do_foo(self, line):
...         print("Hai digitato foo?")
...         print("Linea completa.", line)
...     def do_moo(self, line):
...         print('mooolooooooooo')
...     def default(self, line):
```

```

... print('Comportamento di default')
...
>>> f = Foo()
>>> f.command('foo a b c')
Hai digitato foo?
Linea completa: foo a b c
>>> f.command('moo d e f 33')
mooooooooooooo
>>> f.command('python 3')
Comportamento di default

```

Praticamente in `Cmd.cmdloop()` la classe base (`cmd.Cmd`) delega le azioni da compiere al metodo della classe derivata (`PyWayShell`). La classe `cmd.Cmd` è quindi un framework, che ci consente di scrivere in modo semplice interpreti di comandi.

Infine, osserviamo come nella clausola `except` generica abbiamo messo a log tutti gli errori imprevisti, utilizzando il modulo `logging` della libreria standard.

Per approfondire l'argomento, possiamo consultare la documentazione ufficiale del modulo `cmd`, disponibile alla pagina web <http://docs.python.org/3/library/cmd.html>, e quella del modulo `logging`, alla pagina <http://docs.python.org/3/library/logging.html>. Del modulo `logging` è disponibile anche un *howto*, alla pagina <http://docs.python.org/3/howto/logging.html>.

Attributi magici, metaclassi e test driven development

Questo capitolo finale è dedicato ad alcuni argomenti avanzati: il **modello a oggetti** di Python, le **metaclassi**, gli **attributi magici**, i **descriptor** e il **test driven development**. Nel consueto esercizio conclusivo metteremo in pratica e approfondiremo quanto appreso nel corso del capitolo.

Il modello a oggetti di Python

Abbiamo sempre detto che ogni cosa in Python è un oggetto, per cui le istanze e le classi sono anch'esse degli oggetti. Ma cosa significa, di preciso, questa affermazione? Che una parte degli oggetti sono istanze e un'altra parte no? A quale categoria di oggetti appartengono le funzioni, le liste e i moduli? Sono istanze, classi o nessuna delle due? La risposta è incredibilmente semplice: ogni cosa a cui una etichetta fa riferimento è una istanza, per cui i termini oggetto e istanza sono dei sinonimi.

Questa affermazione sicuramente non risolve tutte le nostre perplessità. Infatti, se ogni oggetto è una istanza, allora anche una classe è una istanza... Un modulo è una istanza, una funzione è una istanza... Ma istanze di cosa? Quale classe crea queste istanze?

La classe `type`

Vediamo di scoprire quale classe crea le classi, ovvero di quale classe sono istanza le classi:

```
>>> class Foo:
...     pass
...
>>> type(Foo)
<class 'type'>
```

Le classi definite dall'utente sono, quindi, istanze della classe `type`:

```
>>> isinstance(Foo, type)
True
```

Non solo, anche le classi built-in sono istanze di `type`:

```
>>> isinstance(list, type)
True
```

Fermiamoci qui con la ricerca, anticipando il risultato: tutte le classi sono istanze della classe `type`. Più avanti, quando parleremo delle metaclassi, vedremo nel dettaglio in cosa consiste il meccanismo di creazione delle classi.

NOTA

Il modulo `types` della libreria standard raccoglie le varie classi, come ad esempio la classe `types.ModuleType` che istanzia i moduli, la classe `types.FunctionType` che istanzia le funzioni ecc.

Se la classe `type` è una classe, in quanto tale deve essere istanza di `type`, ovvero istanza di se stessa. Infatti è proprio così:

```
>>> isinstance(type, type)
True
```

Il cerchio ora è chiuso. Non ci siamo mai accorti di tutto ciò perché la classe `type` agisce in silenzio, per cui istanzia le classi in modo automatico quando viene eseguita l'istruzione `class`. Capiremo meglio questo meccanismo più avanti, quando parleremo delle metaclassi e vedremo che `type` è la metaclassa di default. Anticipiamo che `type` può essere usata anche per creare in modo dinamico nuove classi, come indicato nella docstring:

```
>>> print(type.__doc__)
type(object) -> the object's type
type(name, bases, dict) -> a new type
```

Quando chiamiamo `type` passandole un solo argomento `obj`, allora `type` restituisce `obj.__class__`:

```
>>> t = (1, 2)
>>> t.__class__
<class 'tuple'>
>>> t.__class__ is type(t)
True
>>> mytype = type(t)
>>> mytype(['a', 'b'])
('a', 'b')
```

Quando, invece, viene chiamata passandole tre argomenti, rispettivamente un *nome*, una tupla di *classi base* e un *dizionario* degli attributi, istanzia un nuovo tipo:

```
>>> class A1:
...     a1 = 1
...
>>> class A2:
```

```
... a2 = 2
...
>>> B = type('B', (A1, A2), {'b': 'python'})
```

Il primo argomento viene assegnato al nome della classe:

```
>>> B.__name__
'B'
```

Il secondo indica le classi base:

```
>>> B.__bases__
(<class '__main__.A1'>, <class '__main__.A2'>)
```

Il terzo il dizionario degli attributi:

```
>>> B.b
'python'
```

Possiamo riassumere il tutto nel modo seguente:

- un oggetto è una istanza di una classe;
- le classi sono istanze di una classe, precisamente della classe `type`;
- la classe `type` è una istanza di se stessa;
- i termini oggetto e istanza sono sinonimi.

A questo punto siamo in grado di dare e comprendere la più concisa e precisa definizione di ciò che in Python è un oggetto: *un oggetto è una istanza di una classe*. Visto che le classi sono istanze di una classe, così come i moduli, i package, le funzioni e tutto il resto, la precedente definizione ci riporta a ciò che abbiamo sempre detto, ovvero che *ogni etichetta in Python fa riferimento a un oggetto*.

La classe object

Abbiamo appena visto che tutte le classi sono istanze della classe `type`. Solo le classi sono istanze di `type`, mentre gli altri oggetti non lo sono:

```
>>> isinstance([], type)
False
```

Tutti gli oggetti, però, sono istanze della classe `object`, comprese le classi e quindi `object` stessa:


```
>>> isinstance([], object)
True
>>> isinstance(type, object)
True
>>> isinstance(object, object)
True
```

La classe `object` è, per l'appunto, una classe, per cui è istanza di `type`:

```
>>> isinstance(object, type)
True
```

La classe `object` si trova in cima alla gerarchia di ereditarietà degli oggetti di Python. Tutte le classi, eccetto la classe `object`, hanno `object` come classe base:

```
>>> class Foo:
...     pass
...
>>> Foo.__bases__
(<class 'object'>,)
>>> type.__bases__
(<class 'object'>,)
>>> object.__bases__
()
```

Questo spiega il motivo per cui ogni oggetto è istanza di `object`. Infatti tutte le istanze di una classe sono anche istanze delle classi base di questa:

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(B):
...     pass
...
>>> isinstance(C(), A)
True
```

NOTA

In Python 2.2 il modello a oggetti è stato completamente rivisto. Python 2.2 ha introdotto il concetto di *new-style classes*, ovvero classi equivalenti a quelle di Python 3. Per default in Python 2 le classi sono *old-style*, come

evidenzia il seguente esempio in Python 2.7:

```
>>> class Foo:
...     pass
...
>>> isinstance(Foo, object)
False
```

Per poter creare delle classi new-style, si deve ereditare in modo esplicito da `object`:

```
>>> class Foo(object):
...     pass
...
>>> isinstance(Foo, object)
True
```

Alla pagina web <http://python-history.blogspot.it/2010/06/new-style-classes.html>, Guido riporta vari aneddoti sulla nascita delle classi di nuovo stile.

Tipi e non-tipi

Possiamo dare una nuova definizione di oggetto, equivalente alle precedenti: un oggetto è una istanza della classe `object`. Questa definizione, in accordo con tutte le altre che abbiamo dato, implica che ogni etichetta in Python fa riferimento a un oggetto, visto che `isinstance(etichetta, object)` restituisce sempre `True`:

```
>>> class Foo:
...     pass
...
>>> isinstance(Foo, object)
True
>>> import math
>>> isinstance(math, object)
True
>>> def foo():
...     pass
...
>>> isinstance(foo, object)
True
>>> isinstance(11, object)
True
...
```

L'unica distinzione che ha senso fare è tra oggetti che sono classi e oggetti che

non lo sono, ovvero tra tipi e non-tipi. I tipi sono istanze di `type`, mentre i non-tipi non lo sono. Ecco una serie di oggetti che sono tipi:

```
>>> class Foo:
...     pass
...
>>> isinstance(Foo, type)
True
>>> isinstance(type, type)
True
>>> isinstance(dict, type)
True
>>> isinstance(list, type)
True
>>> isinstance(float, type)
True
>>> isinstance(object, type)
True
```

Ecco una serie di oggetti che non sono tipi:

```
>>> isinstance(Foo(), type)
False
>>> isinstance(object(), type)
False
>>> isinstance(dict(), type)
False
>>> isinstance(list(), type)
False
>>> isinstance(float(), type)
False
```

Più tardi vedremo che le classi le cui istanze sono, a loro volta, delle classi, sono dette *metaclassi*; ciò significa che le istanze delle metaclassi sono, a loro volta, istanze di `type`. La classe `type` stessa è una metaclassa e infatti le sue istanze sono dei tipi:

```
>>> isinstance(type(Foo()), type)
True
```

Anticipiamo che una classe è anche una metaclassa se e solo se è sottoclasse di `type`. Ad esempio, la seguente classe `Foo` non è una metaclassa, mentre `type` sì:

```
>>> issubclass(Foo, type)
False
>>> issubclass(type, type)
True
```

Quindi, riassumendo:

- se un oggetto `obj` non è istanza di `type`, ovvero `isinstance(obj, type)` restituisce `False`, allora `obj` non è una classe;
- se un oggetto `obj` è una istanza di `type`, ovvero se `isinstance(obj, type)` restituisce `True`, allora `obj` è una classe;
- se un oggetto `obj` è una classe e risulta essere sottoclasse di `type`, ovvero `issubclass(obj, type)` restituisce `True`, allora `obj` è una metaclass e questo significa che le istanze di `obj` sono, a loro volta, delle classi.

La parte finale del capitolo è interamente dedicata alle metaclassi, per cui avremo modo di vedere in dettaglio e di approfondire questi concetti.

Duck typing

In Python il tipo di un oggetto è molto importante perché ne determina l'interfaccia, ovvero le azioni che l'oggetto può compiere e il modo in cui interagisce con oggetti di altro tipo:

```
>>> "python" + "3" # Due oggetti di tipo `str` possono essere concatenati
'python3'
>>> "python" + 3 # Un oggetto di tipo `str` non può essere sommato a uno di tipo `int`
Traceback (most recent call last):
...
TypeError: Can't convert 'int' object to str implicitly
```

Per questo motivo talvolta si dice che Python è un linguaggio *fortemente tipizzato (strongly typed)*.

Le etichette, invece, non hanno alcun tipo, poiché sono dei semplici identificativi con i quali fare riferimento agli oggetti. Come sappiamo, infatti, possiamo definire qualsiasi operazione tra le etichette, e solamente a runtime, quando queste verranno sostituite con gli oggetti ai quali fanno riferimento, si valuteranno le operazioni:

```
>>> def itemize(obj):
...     for index, item in enumerate(obj):
...         print(index, '-->', item)
...
>>> itemize(['primo', 44, (1, 2)])
0 --> primo
1 --> 44
2 --> (1, 2)
>>> itemize('ciao')
0 --> c
1 --> i
```

```
2 --> a
3 --> o
```

Questa caratteristica è detta *tipizzazione dinamica* (*dynamic typing*).

La tipizzazione forte non implica assolutamente che per operare sugli oggetti sia necessario conoscerne il tipo, anzi, è buona pratica fare il contrario. La tipizzazione dinamica, infatti, permette di utilizzare uno stile di programmazione detto *duck typing*, che si riassume così: “If it looks like a duck and quacks like a duck, it must be a duck”. Ciò significa che non dobbiamo preoccuparci di verificare che `obj` sia una stringa o una lista, perché magari è un oggetto di un altro tipo ma si comporta ugualmente come ci aspettiamo:

```
>>> itemize({1, 2, 'tre'})
0 --> 1
1 --> 2
2 --> tre
>>> itemize(dict(a=1, b=2, c=3))
0 --> a
1 --> c
2 --> b
>>> open('myfile', 'w').writelines(['prima\n', 'seconda'])
>>> itemize(open('myfile'))
0 --> prima

1 --> seconda
```

Se poi non si comporta come pensavamo, allora possiamo eventualmente gestire l’eccezione.

Quindi ciò che dobbiamo fare è usare le etichette nelle espressioni senza preoccuparci del tipo di oggetto al quale faranno riferimento a runtime. Se proprio non possiamo fare a meno di verificare il tipo, allora utilizziamo `isinstance()` piuttosto che `type()`, poiché, come abbiamo detto nel Capitolo 5, quest’ultima non fa una ricerca lungo la gerarchia di ereditarietà:

```
>>> class Person:
...     def greetings(self):
...         print('hello!')
...
>>> class Student(Person):
...     pass
...
>>> p = Person()
>>> s = Student()
>>> p.greetings()
```

```

hello!
>>> s.greetings()
hello!
>>> if type(s) == Person:
... s.greetings()
...
>>> if isinstance(s, Person):
... s.greetings()
...
hello!

```

Uno studente è una persona, ma se usiamo `type()` non viene riconosciuto come tale.

Overloading dinamico

In vari linguaggi di programmazione è possibile definire funzioni con lo stesso nome, che differiscono per il tipo o il numero dei loro parametri. In questo caso si dice che funzioni omonime sono in *overloading*. In Python, invece, se definiamo due funzioni diverse ma con lo stesso nome, l'ultima definizione sovrascrive le precedenti:

```

>>> def foo(x, y):
... print(x, y)
...
>>> def foo(z):
... print('z:', z)
...
>>> foo(1) # Viene eseguita l'ultima funzione definita
z: 1

```

A partire da Python 3.4, è però possibile effettuare un *overloading dinamico* di una funzione, sulla base del tipo del primo argomento. Per fare ciò si decora la funzione con `functools singledispatch()`:

```

>>> from functools import singledispatch
>>> @singledispatch
... def foo(argA, argB):
... print('Argomenti:', argA, argB, sep=', ')

```

dopodiché si *registrano* le funzioni che si vogliono avere in overloading:

```

>>> @foo.register(int)
... def _(argA, argB):
... print('Primo argomento di tipo int:', argA)

```

```
... print('Secondo argomento:', argB)
...
>>> @foo.register(tuple)
... def _(argA, argB):
...     print('Primo argomento di tipo tuple:', argA)
...     print('Secondo argomento:', argB)
```

Se il primo argomento di `foo()` è una istanza del tipo `int`, allora viene chiamata la funzione decorata con `foo.register(int)`:

```
>>> foo(100, 'python')
Primo argomento di tipo int: 100
Secondo argomento: python
```

Se il primo argomento di `foo()` è una istanza del tipo `tuple`, allora viene chiamata la funzione decorata con `foo.register(tuple)`:

```
>>> foo((1, 2, 3), 'python')
Primo argomento di tipo tuple: (1, 2, 3)
Secondo argomento: python
```

Se il primo argomento di `foo()` non è una istanza né del tipo `int` né del tipo `tuple`, allora viene chiamata la funzione originale, alla quale viene associato il tipo `object`:

```
>>> foo(44.5, 'ciao')
Argomenti: 44.5 ciao
```

Quando la funzione da chiamare viene scelta sulla base del tipo di uno solo dei suoi argomenti, come nel nostro caso, l'algoritmo di selezione viene detto *single dispatch*.

Si osservi che il confronto non viene fatto verificando che il tipo dell'argomento coincida con quello registrato, ma giustamente verificando che il tipo dell'argomento sia istanza di quello registrato:

```
>>> class MyInt(int):
...     pass
...
>>> foo(MyInt(33), 'python')
Primo argomento di tipo int: 33
Secondo argomento: python
```

Se vogliamo associare più di un tipo a una certa implementazione, dobbiamo

decorare la funzione più volte, in cascata:

```
>>> @foo.register(float)
... @foo.register(complex)
... def _(argA, argB):
...     print(argA, "è certamente di tipo float o complex")
...     print(argB, "è di tipo", type(argB))
...
>>> foo(44.5, 'ciao')
44.5 è certamente di tipo float o complex
ciao è di tipo <class 'str'>
>>> foo(1 + 2j, [1, 2])
(1+2j) è certamente di tipo float o complex
[1, 2] è di tipo <class 'list'>
```

Se vogliamo registrare una funzione preesistente, la passiamo come argomento a `foo.register()`:

```
>>> def moo(argA, argB):
...     print('Moooloo: ', argA, argB)
...
>>> foo.register(str, moo)
<function moo at 0x7fdf3cf4b8c8>
>>> foo('python', 'ciao')
Moooloo: python ciao
```

L'attributo `register()` restituisce la funzione originale:

```
>>> fun = foo.register(str, moo)
>>> fun('python', 'ciao')
Moooloo: python ciao
>>> moo is fun
True
```

Consente, inoltre, di registrare le funzioni anonime:

```
>>> foo.register(dict, lambda x, y: print('Primo argomento di tipo dict', x, y))
<function <lambda> at 0x7fdf3cf4b950>
>>> foo({1: 'uno', 2: 'due'}, 'ciao')
Primo argomento di tipo dict {1: 'uno', 2: 'due'} ciao
```

L'attributo `dispatch()` consente di ottenere la funzione associata a un certo tipo:

```
>>> fun = foo.dispatch(str)
>>> fun(100, 'ciao') # Chiama la funzione associata a str
Moooloo: 100 ciao
>>> fun = foo.dispatch(float)
>>> fun(100, 'ciao') # Chiama la funzione associata a float
```


100 è certamente un float o un complex
ciao è di tipo <class 'str'>

Se `dispatch()` non trova l'implementazione associata a quel tipo, restituisce quella associata a `object`:

```
>>> from types import ModuleType
>>> fun = foo.dispatch(ModuleType)
>>> fun('python', 'ciao') # Chiama la funzione associata a object
Argomenti: python ciao
```

L'associazione tipo-funzione viene ottenuta tramite l'attributo `registry`, il quale è un oggetto di tipo mappatura che ha come chiavi i tipi e come valori le corrispondenti implementazioni:

```
>>> for type_, fun in foo.registry.items():
...     print(type_, fun, sep=' -> ')
...
<class 'str'> -> <function moo at 0x7fdf3cf4b8c8>
<class 'dict'> -> <function <lambda> at 0x7fdf3cf4b950>
<class 'float'> -> <function _ at 0x7fdf3cf4b9d8>
<class 'object'> -> <function foo at 0x7fdf3e6e8d90>
<class 'complex'> -> <function _ at 0x7fdf3cf4b9d8>
<class 'int'> -> <function _ at 0x7fdf3cf4b840>
```

Quindi un altro modo per ottenere una particolare implementazione è quello di utilizzare l'attributo `registry`:

```
>>> fun = foo.registry[int]
>>> fun('python', 'ciao') # Non chiama l'implementazione associata a str
Primo argomento di tipo int: python
Secondo argomento: ciao
>>> foo.registry[int] is foo.dispatch(int)
True
```

L'overloading dinamico è stato introdotto con la PEP-0443.

NOTA

Ricordiamo, inoltre, che esiste una variante di Python, chiamata *mypy*, che fa uso delle function annotation per combinare assieme i vantaggi della tipizzazione statica e dinamica. Si veda, a tale proposito, la nota nella sezione *Introspezione con le annotazioni* del [Capitolo 3](#).

Gli attributi magici

Python attribuisce un significato speciale ad alcuni attributi che iniziano e finiscono con due underscore, i quali vengono detti *attributi magici* o anche *attributi speciali*. Alcuni di essi sono definiti nella classe `object`, come, ad esempio, l'inizializzatore:

```
>>> hasattr(object, '__init__') # Tutti gli oggetti ereditano l'inizializzatore da `object`
True
```

Visto che la classe `object` si trova al vertice della gerarchia di ereditarietà, i suoi attributi vengono ereditati da tutte le classi e quindi appartengono a tutti gli oggetti:

```
>>> class Foo:
...     pass
...
>>> for attr_name in dir(object):
...     if not hasattr(Foo(), attr_name):
...         print("L'istanza Foo() di Foo non ha l'attributo ", attr_name)
...
>>>
```

Ognuno degli attributi speciali svolge un compito preciso. Ad esempio, il metodo speciale `__new__()` si occupa della creazione dell'istanza, `__init__()` della sua inizializzazione ecc.

La classe `object`, o, più in generale, la classe base, non fa altro che definire delle azioni di default da compiere quando le classi derivate non fanno l'overriding dei metodi speciali. Quindi, se vogliamo personalizzare i compiti assegnati ai metodi magici, dobbiamo farne l'overriding, implementando il comportamento che ci interessa.

Vi sono metodi speciali per i quali non sono previste delle azioni di default. Questi metodi, pertanto, non sono definiti in `object` e quindi nessun oggetto li ha. Uno di questi, ad esempio, è il distruttore:

```
>>> hasattr(object, '__del__'), hasattr(Foo(), '__del__')
(False, False)
```

Questi sono semplicemente dei metodi che Python considera speciali e che chiama all'occorrenza se il programmatore li definisce. Metodi di questo tipo sono, ad esempio, `__iter__()` e `__contains__()`, che abbiamo introdotto nella sezione *Esempio di utilizzo di un metodo statico* del [Capitolo 5](#).

La creazione delle istanze

In questa sezione ci occuperemo della fase di *creazione* delle istanze, la quale coinvolge i metodi speciali `__call__()`, `__new__()` e `__init__()`.

Il metodo speciale `__call__()`

Quando una classe `Foo` definisce il metodo `__call__()`, allora le istanze di `Foo` diventano oggetti *callable* e una loro chiamata corrisponde alla chiamata al metodo `Foo.__call__()`:

```
>>> class Foo:
...     pass
...
>>> f = Foo()
>>> callable(f)
False
>>> class Foo:
...     def __call__(self, a, b):
...         print(a, b)
...
>>> f = Foo()
>>> callable(f)
True
>>> f('python', 3) # Viene chiamato il metodo __call__()
python 3
```

Come sappiamo, le classi sono istanze e per default sono istanziate direttamente dalla classe `type`, per cui la chiamata `Foo()` corrisponde alla chiamata al metodo `type(Foo).__call__()`, **OVVERO** `type.__call__()`.

NOTA

Più avanti, quando parleremo delle metaclassi, ci occuperemo in dettaglio della chiamata al metodo `type.__call__()`.

La chiamata a `type.__call__()` gestisce la creazione e l'inizializzazione dell'istanza. Per fare ciò il metodo di classe `type.__call__()` chiama prima il metodo statico

`Foo.__new__()`, il quale crea l'istanza e la restituisce. A questo punto, se indichiamo con `inst` l'istanza restituita da `Foo.__new__()`, `type.__call__()` verifica che `isinstance(inst, Foo)` sia `True` e, in questo caso, chiama `inst.__init__()` in modo da inicializzarla e poi restituisce l'istanza. Il tutto può essere riassunto nel modo seguente:

```
class type:
    @classmethod
    def __call__(cls, *args, **kwargs):
        inst = cls.__new__(cls, *args, **kwargs)
        if isinstance(inst, cls):
            inst.__init__(*args, **kwargs)
        return inst
```

Il metodo speciale `__new__()`

Tutte le classi hanno il metodo speciale `__new__()`. Questo è un metodo statico definito nella classe `object` e quindi presente in tutte le classi. La chiamata a una classe `Foo`, come abbiamo detto, causa la chiamata al metodo `type.__call__()`, il quale si occupa di chiamare `Foo.__new__()`. Quest'ultimo *crea* una istanza di `Foo` e la restituisce:

```
>>> f = Foo.__new__(Foo)
>>> isinstance(f, Foo)
True
```

Per questo motivo il metodo `__new__()` è detto *costruttore*. Nella classe `Foo` non è stato definito il metodo `Foo.__new__()`, per cui, quando questo viene chiamato da `type.__call__()`, si risale la gerarchia di ereditarietà seguendo il MRO e quindi viene fatta la chiamata al corrispondente metodo della classe base, ovvero `object.__new__(Foo)`.

Per personalizzare la creazione delle istanze dobbiamo quindi fare l'overriding del metodo `__new__()`. Quando `__call__()` esegue la chiamata a `__new__()`, gli passa la classe come primo argomento, per cui dobbiamo necessariamente indicarlo come parametro nella definizione:

```
>>> class Foo:
...     def __new__(cls):
...         print('Foo.__new__()')
...         return super().__new__(cls)
...
>>> f = Foo()
Foo.__new__()
```

Dobbiamo ricordarci di delegare la creazione dell'istanza al costruttore della classe base, altrimenti la chiamata diretta a `cls()` nell'istruzione `return` darebbe luogo a una ricorsione infinita, visto che `cls()` chiamerebbe nuovamente `Foo.__new__()`:

```
>>> class Foo:
...     def __new__(cls):
...         print('Foo.__new__()')
...         return cls()
...
>>> f = Foo()
Foo.__new__()
Foo.__new__()
Foo.__new__()
...
RuntimeError: maximum recursion depth exceeded while calling a Python object
```

Se `Foo.__new__()` restituisce una istanza `f` di `Foo`, e questa ha l'attributo `__init__()`, allora `type.__call__()` chiama `f.__init__()`:

```
>>> class Foo:
...     def __new__(cls):
...         print('Foo.__new__()')
...         return super().__new__(cls)
...     def __init__(self):
...         print('Foo.__init__()')
...
>>> f = Foo()
Foo.__new__()
Foo.__init__()
```

Se, invece, `Foo.__new__()` non restituisce una istanza di `Foo`, allora il metodo `type.__call__()`, non avendo il riferimento all'istanza, non può chiamare il metodo `__init__()` di questa per inizializzarla:

```
>>> class Foo:
...     def __new__(cls):
...         print('Foo.__new__()')
...         def __init__(self):
...             print('Foo.__init__()')
...
>>> f = Foo()
Foo.__new__()
```

Inoltre `__new__()` in questo caso restituisce implicitamente `None`, per cui anche `type.__call__()` restituisce `None` e l'etichetta `f` fa quindi riferimento a `None`:

```
>>> type(f)
<class 'NoneType'>
```

Se la classe va chiamata passandole degli argomenti, allora è necessario indicare i corrispondenti parametri nella definizione di `__new__()`:

```
>>> class Foo:
...     def __new__(cls, *args, **kwargs):
...         for arg in args:
...             print(arg)
...         for k, v in kwargs.items():
...             print(k, v)
...         return super().__new__(cls)
...
>>> f = Foo()
>>> f = Foo(1, 2)
1
2
>>> f = Foo(1, 2, a=3)
1
2
a 3
```

Se non lo si fa, allora si ottiene un errore, visto che `type.__call__()` chiama `__new__()` passandole gli argomenti passati alla classe al momento della chiamata:

```
>>> class Foo:
...     def __new__(cls):
...         print('__new__()')
...     def __init__(self, a):
...         print(a)
...
>>> f = Foo(3)
Traceback (most recent call last):
...
TypeError: __new__() takes 1 positional argument but 2 were given
```

Anche se solitamente non si ha l'esigenza di fare l'overriding di `__new__()`, vi sono dei casi in cui è necessario farlo (come vedremo, ad esempio, nell'esercizio pratico sul test driven development) e altri in cui può essere utile farlo. Vediamone qualcuno.

Implementazione del singleton pattern con overriding del costruttore

Il *singleton pattern* ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza. In Python una possibile implementazione del singleton pattern consiste nel fare l'overriding del metodo `__new__()`, in modo che questo crei una istanza se non ne è già stata creata una, altrimenti restituisca l'istanza precedentemente creata:

```
>>> class Foo:
...     def __new__(cls):
...         if not hasattr(cls, '_instance'):
...             cls._instance = super().__new__(cls)
...         return cls._instance
...
>>> f1 = Foo()
>>> f2 = Foo()
>>> f1 is f2
True
```

Si osservi che questa implementazione fa sì che l'istanza venga creata una sola volta, ma non impedisce che venga inizializzata più volte:

```
>>> class Foo:
...     def __new__(cls, *args, **kwargs):
...         if not hasattr(cls, '_instance'):
...             cls._instance = super().__new__(cls)
...         return cls._instance
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
...
>>> f1 = Foo(1, 2)
>>> f1.a
1
>>> f1.b
2
>>> f2 = Foo(3, 4)
>>> f1.a
3
>>> f1.b
4
```

Anche se non definiamo l'inizializzatore nella classe che implementa il singleton, si ha ugualmente inizializzazione multipla per le istanze delle classi derivate che fanno l'overriding di `__init__()`:

```

>>> class Foo:
...     def __new__(cls, *args, **kwargs):
...         if not hasattr(cls, '_instance'):
...             cls._instance = super().__new__(cls)
...         return cls._instance
...
>>> class A(Foo):
...     def __init__(self, a, b):
...         self.a = a
...         self.b = b
...
>>> a1 = A(1, 2)
>>> a1.a, a1.b
(1, 2)
>>> a2 = A(3, 4)
>>> a1.a, a1.b
(3, 4)
>>> a1 is a2
True

```

NOTA

Tipicamente, quando si chiama una classe ci si aspetta che questa crei una nuova istanza e la restituisca. Per questo motivo alcuni programmatori pensano che questa implementazione del singleton pattern violi il *principio di minima sorpresa (POLS: principle of least surprise)*, secondo il quale il comportamento dell'oggetto dovrebbe essere quello che sorprende meno l'utilizzatore che non conosce i suoi dettagli implementativi.

Ad esempio, la seguente è una implementazione del singleton pattern più esplicita della precedente:

```

>>> class Foo:
...     @classmethod
...     def singleton(cls):
...         if not hasattr(cls, '_instance'):
...             cls._instance = cls()
...         return cls._instance

```

In questo caso per ottenere un singleton dobbiamo chiamare esplicitamente `Foo.singleton()`:

```

>>> f1 = Foo.singleton()

```



```
>>> f2 = Foo.singleton()
>>> f1 is f2
True
```

Vi sono tante altre possibili implementazioni del singleton pattern in Python, ma non ci dilunghiamo ulteriormente, il nostro scopo è solamente quello di mostrare un esempio di overriding del metodo `__new__()`.

Ereditare dai tipi immutabili

Supponiamo di voler creare un nuovo tipo di lista:

```
>>> class FooList(list):
...     def __init__(self, seq):
...         result = list(seq) + [sum(seq)]
...         super().__init__(result)
```

La classe `FooList` prende una sequenza di numeri e istanzia una lista che ha gli stessi elementi della sequenza, più un altro elemento che è la somma dei precedenti:

```
>>> mylist = FooList((1, 2, 3))
>>> mylist
[1, 2, 3, 6]
```

Poiché abbiamo ereditato da `list`, la nostra istanza ha tutti i metodi delle liste:

```
>>> mylist.append(12)
>>> mylist
[1, 2, 3, 6, 12]
>>> mylist.count(3)
1
```

Questa classe non è molto utile, ma ci consente di capire l'utilità del metodo `__new__()` quando vogliamo ereditare dai tipi immutabili come `str`, `tuple`, `int` ecc.

Supponiamo, infatti, di voler creare una classe analoga a `FooList`, che però istanzi delle `tuple`:

```
>>> class FooTuple(tuple):
...     def __init__(self, seq):
...         result = tuple(seq) + (sum(seq),)
...         super().__init__(result)
... 
```

```
>>> mytuple = FooTuple((1, 2, 3))
Traceback (most recent call last):
...
TypeError: object.__init__() takes no parameters
```

Vediamo di capire cos'è successo. Come sappiamo, le tuple sono oggetti immutabili, per cui, una volta create, non possono essere modificate. Non serve, quindi, avere un inizializzatore, e infatti i tipi immutabili non effettuano l'overriding di `__init__()`, ma ereditano il metodo della classe base:

```
>>> float.__init__ is object.__init__
True
>>> tuple.__init__ is object.__init__
True
```

Questo non accade per i tipi degli oggetti mutabili, i quali effettuano l'overriding di `__init__()`:

```
>>> list.__init__ is object.__init__
False
>>> dict.__init__ is object.__init__
False
```

Quindi, se ereditiamo dalla classe `tuple`, allora `super().__init__()` equivale a `object.__init__()` e quest'ultimo non accetta parametri, da cui l'errore.

La classe `tuple`, però, effettua l'overriding del metodo `object.__new__()`:

```
>>> float.__new__ is object.__new__
False
```

La soluzione, quindi, consiste nel fare l'overriding del costruttore:

```
>>> class FooTuple(tuple):
...     def __new__(cls, seq):
...         result = tuple(seq) + (sum(seq),)
...         return super().__new__(cls, result)
...
>>> mytuple = FooTuple((1, 2, 3))
>>> mytuple
(1, 2, 3, 6)
```

NOTA

In Python 2 i metodi `object.__init__()` e `object.__new__()` accettano degli argomenti, ma

questi vengono scartati. Se, infatti, eseguiamo l'esempio precedente con Python 2.7, non otteniamo alcun errore:

```
>>> class FooTuple(tuple):
...     def __init__(self, seq):
...         result = tuple(seq) + (sum(seq),)
...         super(tuple, self).__init__(result)
...
>>> FooTuple((1, 2, 3))
(1, 2, 3)
```

Sarebbe stato meglio se avessimo ottenuto un errore, perché la classe non funziona, visto che non viene aggiunto un elemento pari alla somma dei precedenti. Infatti in questo caso il metodo `type.__call__()` passa la sequenza al metodo `tuple.__new__()`, il quale crea la tupla. Poi viene chiamato il metodo `tuple.__init__()`, OVVERO `object.__init__()`, al quale viene passato `result`. Questo argomento viene scartato, per cui il metodo non esegue alcuna azione utile.

Il metodo speciale `__init__()`

Dopo che `type.__call__()` ha chiamato il metodo statico `__new__()` della classe, se questo restituisce una istanza della classe, allora `type.__call__()` provvede a chiamare l'inizializzatore `__init__()` di tale istanza.

Anche l'inizializzatore è definito nella classe `object`, per cui è presente in tutte le classi.

Se facciamo l'overriding dell'inizializzatore, dobbiamo sapere come gestire i suoi parametri e il valore di ritorno. Innanzitutto, l'inizializzatore deve restituire `None`:

```
>>> class Foo:
...     def __init__(self):
...         return True
...
>>> Foo()
Traceback (most recent call last):
...
TypeError: __init__() should return None, not 'bool'
```

Come sappiamo, se non inseriamo l'istruzione `return`, allora la funzione restituisce implicitamente `None`, e questo è ciò che faremo.

Per quanto riguarda i parametri, il primo, come sappiamo, viene chiamato usualmente `self`; a esso viene assegnata l'istanza, mentre gli altri sono quelli che

vengono associati agli argomenti passati alla classe al momento della chiamata:

```
>>> class Foo:
...     def __init__(self, a, b):
...         print(a, b)
...
>>> f = Foo('python', 3)
python 3
```

Se in una classe derivata facciamo l'overriding dell'inizializzatore, dobbiamo assicurarci di chiamare esplicitamente l'inizializzatore della classe base:

```
>>> class A:
...     def __init__(self, name):
...         self.name = name
...
>>> class B(A):
...     def __init__(self, name, nickname):
...         self.nickname = nickname
...         super().__init__(name) # Inizializziamo la classe base
...
>>> b = B('Gigi Riva', 'rombo di tuono')
>>> b.name
'Gigi Riva'
>>> b.nickname
'rombo di tuono'
```

La finalizzazione delle istanze

Per personalizzare la fase di distruzione di un oggetto dobbiamo definire il metodo `__del__()`, comunemente detto *distruttore*. La classe `object` non lo definisce, per cui le classi per default non lo hanno:

```
>>> class Foo:
...     pass
...
>>> hasattr(Foo, '__del__')
False
```

Il distruttore viene chiamato prima che venga liberata la memoria dall'istanza:

```
>>> class Foo:
...     def __new__(cls):
...         print('Sto creando una istanza di Foo')
```

```

... return super().__new__(cls)
... def __init__(self):
...     print("Sto inizializzando l'istanza creata")
... def __del__(self):
...     print("Sto eseguendo qualche operazione, prima che l'istanza venga distrutta")
...
>>> f = Foo()
Sto creando una istanza di Foo
Sto inizializzando l'istanza creata
>>> del f
Sto eseguendo qualche operazione, prima che l'istanza venga distrutta

```

Questo esempio potrebbe trarci in inganno: il distruttore di un oggetto non viene chiamato quando si cancella una etichetta che fa riferimento a esso, ma prima che l'oggetto venga distrutto. L'oggetto viene distrutto quando non vi sono più etichette che fanno riferimento a esso:

```

>>> class Foo:
...     def __del__(self):
...         print(self, ' sta per essere distrutto...')
...
>>> a = Foo()
>>> a = 44 # L'etichetta `a` viene assegnata a un nuovo oggetto
<__main__.Foo object at 0xb72be2ac> sta per essere distrutto...
>>> a = b = Foo() # Due etichette fanno riferimento all'oggetto
>>> del a # Non viene chiamato `__del__` perché l'oggetto non sta per essere distrutto
>>> del b # Adesso viene chiamato `__del__`
<__main__.Foo object at 0xb72be32c> sta per essere distrutto...

```

Se una classe derivata non definisce il metodo `__del__()`, allora, se la classe base lo definisce, questo viene chiamato automaticamente:

```

>>> class A:
...     def __del__(self):
...         print('A.__del__()')
...
>>> class B(A):
...     pass
...
>>> b = B()
>>> del b
A.__del__()

```

Se, invece, facciamo l'overriding del distruttore, allora dobbiamo assicurarci di chiamare esplicitamente il distruttore della classe base, altrimenti

quest'ultimo non verrebbe chiamato:

```
>>> class B(A):
...     def __del__(self):
...         print('B.__del__()')
...
>>> b = B()
>>> del b
B.__del__()
>>> class B(A):
...     def __del__(self):
...         print('B.__del__()')
...         super().__del__()
...
>>> b = B()
>>> del b
B.__del__()
A.__del__()
```

Vi sono dei casi in cui il finalizzatore non viene chiamato in modo automatico, perché si verifica un cosiddetto *reference cycle*, come nell'esempio seguente:

```
>>> class A:
...     def __init__(self, a):
...         self.a = a
...     print('In A.__init__()')
...     def __del__(self):
...         print('Arrivederci da A()')
...
>>> class B:
...     def __init__(self):
...         self.b = A(self) # Reference cycle
...     print('In B.__init__()')
...     def __del__(self):
...         print('Arrivederci da B()')
...
>>> b = B()
In A.__init__()
In B.__init__()
```

Come possiamo vedere, se eliminiamo l'etichetta `b`, il metodo `__del__()` non viene chiamato:

```
>>> del b
>>>
```

A partire da Python 3.4, il meccanismo di finalizzazione è stato migliorato (vedi PEP-0442) e il metodo `gc.collect()` è in grado di liberare la memoria anche in questi casi:

```
>>> import gc # Python 3.4
>>> gc.collect()
Arrivederci da B()
Arrivederci da A()
4
```

Overloading degli operatori

In questa sezione approfondiremo l'*overloading degli operatori*, concetto introdotto nella sezione *Un primo sguardo all'overloading degli operatori* del Capitolo 5. Forniremo una spiegazione esaustiva di come effettuare l'overloading dei principali *operatori aritmetici*: di *somma*, di *sottrazione* e di *moltiplicazione*. L'overloading dei restanti operatori si effettua con i medesimi criteri.

Overloading degli operatori aritmetici ed ereditarietà

Nella sezione introduttiva del Capitolo 5 abbiamo visto come effettuare l'overloading dell'operatore di *somma*, affinché si possano sommare due oggetti di tipo `Circle`. Riportiamo qui la classe, tralasciando di scrivere il metodo `Circle.area()`, in modo da concentrarci unicamente sull'aspetto dell'overloading:

```
>>> class Circle:
...     def __init__(self, radius: float = 1.0):
...         self.radius = radius
...     def __add__(self, other):
...         return Circle(self.radius + other.radius)
...
>>> c1 = Circle()
>>> c2 = Circle(9)
>>> c = c1 + c2
>>> c.radius
10.0
>>> type(c)
<class '__main__.Circle'>
```

Tutto funziona a dovere: la somma viene calcolata correttamente e il risultato, giustamente, è un oggetto di tipo `Circle`.

Supponiamo di voler creare una classe `Wheel` che rappresenti una generica

ruota. Poiché una ruota sostanzialmente è *un* cerchio, facciamo in modo che `Wheel` derivi da `Circle`:

```
>>> class Wheel(Circle):  
... pass
```

La classe `Wheel` eredita tutti gli attributi di `Circle`, quindi anche il metodo `Circle.__add__()`. Possiamo, pertanto, sommare due oggetti di tipo `Wheel`:

```
>>> w1 = Wheel()  
>>> type(w1)  
<class '__main__.Wheel'>  
>>> w1.radius  
1.0  
>>> w2 = Wheel(14)  
>>> w2.radius  
14  
>>> w = w1 + w2  
>>> w.radius  
15.0
```

Benissimo, ancora una volta tutto sembra funzionare come dovrebbe. Non proprio tutto, però... Se, infatti, verifichiamo il tipo di oggetto restituito dalla somma, troviamo una sorpresa:

```
>>> type(w)  
<class '__main__.Circle'>
```

La somma di due ruote dà come risultato un cerchio, e questo non va per nulla bene. Vediamo di capire cosa è successo. Quando Python fa la somma `w1 + w2`, esegue la chiamata `Wheel.__add__(w1, w2)`. La classe `Wheel` non ha definito il suo metodo `Wheel.__add__()`, per cui il metodo che viene chiamato è quello ereditato dalla classe padre, ovvero `Circle.__add__()`:

```
>>> Wheel.__add__  
<function Circle.__add__ at 0xb72dc8e4>
```

Il problema è insito in questo metodo, poiché restituisce un `Circle(self.radius + other.radius)` e non `Wheel(self.radius + other.radius)`. Per fare in modo che `Circle.__add__()` restituisca un oggetto di tipo `Circle` quando si fa la somma di due `Circle` e un oggetto di tipo `Wheel` quando si fa la somma di due `Wheel`, dobbiamo restituire `self.__class__(self.radius + other.radius)`:

```
>>> class Circle:
```



```

... def __init__(self, radius: float = 1.0):
...     self.radius = radius
... def __add__(self, other):
...     return self.__class__(self.radius + other.radius)
...
>>> c1 = Circle()
>>> c2 = Circle(9)
>>> c = c1 + c2
>>> c.radius
10.0
>>> type(c)
<class '__main__.Circle'>
>>> class Wheel(Circle):
...     pass
...
>>> w1 = Wheel(2)
>>> w2 = Wheel(8)
>>> w = w1 + w2
>>> w.radius
10
>>> type(w)
<class '__main__.Wheel'>

```

Questa volta, infatti, quando viene eseguita la somma `w1 + w2`, e quindi chiamato `Circle.__add__(w1, w2)`, viene creato un oggetto `self.__class__(self.radius + other.radius)`. Poiché a `self` viene assegnato `w1`, questo oggetto è `w1.__class__(self.radius + other.radius)`. Se osserviamo che `w1.__class__` è la classe `Wheel`:

```

>>> w1.__class__ is Wheel
True

```

ci rendiamo conto che scrivere `w1.__class__(self.radius + other.radius)` è la stessa identica cosa che scrivere `Wheel(self.radius + other.radius)`, e questo è esattamente ciò che volevamo.

Operazioni commutative

Consideriamo ancora la nostra classe `Circle`. Se diamo uno sguardo più attento alla sua definizione, notiamo che la somma è possibile anche se l'operando di destra non è un oggetto di tipo `Circle`. È sufficiente, infatti, che questo sia un oggetto che ha un attributo `radius` di tipo numerico:

```

>>> class Foo:
...     def __init__(self, num):
...         self.radius = num
...

```

```
>>> c = Circle()
>>> f = Foo(44)
>>> r = c + f
>>> r.radius
45.0
```

La somma, però, non è commutativa, poiché non abbiamo definito il metodo `Foo.__add__()`; ciò significa che se Python deve valutare `f + c`, non potendo chiamare `f.__add__(c)`, non sa come fare la somma:

```
>>> c + f
<__main__.Circle object at 0xb720a2ec>
>>> f + c
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Foo' and 'Circle'
```

Per fare in modo che la somma sia *commutativa*, la classe `Circle` deve definire il metodo `Circle.__radd__()`, dove la lettera *r* sta per *right*. La somma destra di `a + b` viene fatta qualificando il metodo tramite l'operando destro: `b.__radd__(a)`.

Vediamo di chiarire in modo esaustivo come Python esegue la somma `a + b`:

- se esiste il metodo `a.__add__()`, allora viene chiamato `a.__add__(b)`:

```
>>> class A:
...     def __add__(self, other):
...         print('A.__add__()')
...
>>> class B:
...     def __radd__(self, other):
...         print('B.__radd__()')
...
>>> a = A()
>>> b = B()
>>> a + b
A.__add__()
```

- se `a.__add__(b)` restituisce `NotImplemented` ed esiste il metodo `b.__radd__()`, allora viene fatta anche la chiamata `b.__radd__(a)`:

```
>>> class A:
...     def __add__(self, other):
...         print('A.__add__()')
...         return NotImplemented
...
>>> class B:
...     def __radd__(self, other):
...         print('B.__radd__()')
...
... 
```

```
>>> a = A()
>>> b = B()
>>> a + b
A.__add__()
B.__radd__()
```

- se `a.__add__()` non esiste ed esiste `b.__radd__()`, allora viene fatta la chiamata `b.__radd__(a)`:

```
>>> class A:
...     pass
...
>>> class B:
...     def __radd__(self, other):
...         print('B.__radd__()')
...
>>> a = A()
>>> b = B()
>>> a + b
B.__radd__()
```

- se non esiste `a.__add__()` e neppure `b.__radd__()`, viene lanciata una eccezione di tipo `TypeError`:

```
>>> class A:
...     pass
...
>>> class B:
...     pass
...
>>> a = A()
>>> b = B()
>>> a + b
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'A' and 'B'
```

In virtù di quanto abbiamo appena detto, per poter effettuare la somma commutativa tra oggetti di tipo `Circle` e oggetti di tipo `Foo`, possiamo definire il metodo `Circle.__radd__()` nel modo seguente:

```
>>> class Circle:
...     def __init__(self, radius: float = 1.0):
...         self.radius = radius
...     def __add__(self, other):
...         return self.__class__(self.radius + other.radius)
...     def __radd__(self, other):
...         return self.__class__(self.radius + other.radius)
```

```
...
>>> class Foo:
...     def __init__(self, num):
...         self.radius = num
...
>>> c = Circle()
>>> f = Foo(44)
>>> (c + f).radius, (f + c).radius
(45.0, 45.0)
```

La somma `c + f` viene quindi tradotta nella chiamata `c.__add__(f)`, la quale restituisce un oggetto `Circle(c.radius + f.radius)`, ovvero `Circle(1.0 + 44)`. La somma `f + c`, invece, porta alla chiamata `c.__radd__(f)`, nella quale viene assegnata a `self` l'istanza `c` e a `other` quella `f`, per cui viene, ancora una volta, restituito `Circle(c.radius + f.radius)`.

NOTA

Se avessimo definito `Circle.__radd__()` nel modo seguente:

```
def __radd__(self, other):
    return self + other
```

la somma `f + c` avrebbe portato alla chiamata `c.__radd__(f)` e questa avrebbe valutato `self + other`, ovvero `c + f`, la quale avrebbe portato alla chiamata `c.__add__(f)`. Avremmo avuto, quindi, lo stesso risultato, ma con una chiamata a funzione in più, e quindi una operazione di somma più lenta:

```
>>> class Circle1:
...     def __init__(self, radius: float = 1.0):
...         self.radius = radius
...     def __add__(self, other):
...         return self.__class__(self.radius + other.radius)
...     def __radd__(self, other):
...         return self.__class__(self.radius + other.radius)
...
>>> class Circle2:
...     def __init__(self, radius: float = 1.0):
...         self.radius = radius
...     def __add__(self, other):
...         return self.__class__(self.radius + other.radius)
...     def __radd__(self, other):
...         return self + other
```

```
...
>>> c1 = Circle1()
>>> c2 = Circle2()
>>> import timeit
>>> timeit.timeit(stmt='f + c1', setup='from __main__ import f, c1')
1.230488768000214
>>> timeit.timeit(stmt='f + c2', setup='from __main__ import f, c2')
1.6209731959970668
```

Potremmo anche definire la somma tra un numero e un oggetto di tipo `Circle`, in modo che dia come risultato un `Circle` di raggio pari al raggio dell'oggetto sommato al numero:

```
>>> class Circle:
...     def __init__(self, radius: float = 1.0):
...         self.radius = radius
...     def __add__(self, ro):
...         return self.__class__(self.radius + ro.radius \
...             if hasattr(ro, 'radius') else self.radius + ro)
...     def __radd__(self, lo):
...         return self.__class__(self.radius + lo.radius \
...             if hasattr(lo, 'radius') else self.radius + lo)
...
...
>>> c = Circle()
>>> (c + 1).radius
2.0
>>> (1 + c).radius
2.0
>>> (f + c).radius
45.0
>>> (c + f).radius
45.0
```

Gli operatori di augmented assignment

Si può effettuare la somma di oggetti anche tramite l'operatore di *augmented assignment*. Per fare ciò si definisce il comportamento nel metodo `__iadd__()`:

```
>>> class Circle:
...     def __init__(self, radius: float = 1.0):
...         self.radius = radius
...     def __iadd__(self, other):
...         self.radius += other.radius if hasattr(other, 'radius') else other
...         return self
...
...
```

```
>>> c = Circle()
>>> f = Foo(14)
>>> c += f
>>> c.radius
15.0
>>> c += 1
>>> c.radius
16.0
```

NOTA

In questo caso l'*augmented assignment* non modifica l'oggetto a cui `self.radius` fa riferimento, poiché questo, essendo un numero, è immutabile. Si osservi, però, che in generale l'approccio da utilizzare è quello descritto nell'esempio, dove prima viene effettuato l'*augmented assignment* e dopo viene restituita l'istanza. Così facendo, se `self.radius` facesse riferimento a un oggetto mutabile, questo verrebbe giustamente modificato, come indicato nella PEP-0203.

La classe si comporta bene anche quando viene ereditata:

```
>>> class Wheel(Circle):
...     pass
...
>>> w1 = Wheel()
>>> w1 += Wheel(3)
>>> w1.radius
4.0
>>> type(w1)
<class '__main__.Wheel'>
```

Sottrazione, moltiplicazione e valore assoluto

I concetti visti nelle precedenti sezioni sono generali, per cui non ripeteremo la trattazione anche per gli operatori di *sottrazione* e *moltiplicazione*, ma ci limiteremo a mostrare un esempio di utilizzo:

```
# point.py
import math
from collections import namedtuple
class Point(namedtuple('Point', 'x, y')):
    """Un punto geometrico nel piano cartesiano."""
    def __new__(cls, x=0, y=0):
        """Crea e restituisci una istanza di 'Point'."""
        return super().__new__(cls, x, y)
    def __add__(self, other):
        """Restituisci la somma 'a + b': 'a + b -> Point(a[0] + b[0], a[1] + b[1])'."""
        (self_x, self_y), (other_x, other_y) = self, other
        return self.__class__(self_x + other_x, self_y + other_y)
```

```

def __radd__(self, other):
    """Restituisci la somma `obj + p`: `obj + p -> p + obj`."""
    return self + other
def __sub__(self, other):
    """Restituisci la sottrazione `a - b`: `a - b -> Point(a[0] - b[0], a[1] - b[1])`."""
    (self_x, self_y), (other_x, other_y) = self, other
    return self.__class__(self_x - other_x, self_y - other_y)
def __rsub__(self, other):
    """Restituisci la sottrazione `obj - p`: `obj - p -> p - obj`."""
    return self - other
def __mul__(self, other):
    """Restituisci il prodotto `a * b`: `a * b -> Point(a[0] * b[0], a[1] * b[1])`."""
    (self_x, self_y), (other_x, other_y) = self, other
    return self.__class__(self_x * other_x, self_y * other_y)
def __rmul__(self, other):
    """Restituisci il prodotto `obj * p`: `obj * p -> p * obj`."""
    return self * other
def __pos__(self):
    """Restituisci l'operazione unaria `+p`: `+p -> Point(abs(p.x), abs(p.y))`."""
    return self.__class__(abs(self.x), abs(self.y))
def __neg__(self):
    """Restituisci l'operazione unaria `-p`: `-p -> Point(-abs(p.x), -abs(p.y))`."""
    return self.__class__(-abs(self.x), -abs(self.y))
def __invert__(self):
    """Restituisci l'operazione unaria `~p`: `~p -> Point(-p.x, -p.y)`."""
    return self.__class__(-self.x, -self.y)
def __pow__(self, exponent):
    """Elevamento a potenza: `p ** e -> Point(p.x**e, p.y**e)`."""
    return self.__class__(self.x ** exponent, self.y ** exponent)
def __abs__(self):
    """Distanza dall'origine: `abs(p) -> (p.x**2 + p.y**2)**0.5`."""
    return math.sqrt(self.x**2 + self.y**2)
def distance(self, other=(0, 0)):
    """Restituisci la distanza tra due punti.

```

Con un argomento: `p.distance(a) -> ((p.x - a.x)**2 + (p.y - a.y)**2)** 0.5`.`
 Senza un argomento restituisci la distanza dall'origine: `p.distance() -> abs(p)`.`

```

"""
return abs(self - other)
def round(self, ndigits=0):
    """Restituisci un punto con le coordinate round"""
    return self.__class__(round(self.x, ndigits), round(self.y, ndigits))

```

Il metodo speciale `obj.__abs__()` viene chiamato quando si chiama la funzione `abs(obj)`.

NOTA

In realtà una classe punto geometrico dovrebbe restituire come differenza tra due punti un vettore, ma abbiamo preferito commettere questo errore al fine di semplificare la trattazione e di mettere in evidenza solamente l'overloading

Ecco un esempio di utilizzo:

```
Point(x=4, y=15)
>>> from point import *
>>> p = Point(2, 5)
>>> print(p)
Point(x=2, y=5)
>>> p.distance(Point(-1, 1))
5.0
>>> abs(p)
5.385164807134504
>>> p + Point(2, 3)
Point(x=4, y=8)
>>> p * (2, 2)
Point(x=4, y=10)
>>> p - Point(2, 5)
Point(x=0, y=0)
>>> ~p
Point(x=-2, y=-5)
>>> Point(2.63, -5.66).round()
Point(x=3.0, y=-6.0)
```

L'elenco degli operatori, con il corrispondente significato, è riportato nella Tabella 6.1.

Tabella 6.1 - Attributi magici da utilizzare per effettuare l'overloading degli operatori.

Metodo magico	Significato
<code>__add__</code>	+
<code>__sub__</code>	-
<code>__mul__</code>	*
<code>__floordiv__</code>	//
<code>__div__</code>	
<code>__truediv__</code>	/
<code>__mod__</code>	%
<code>__divmod__</code>	divmod(a, b)
<code>__pow__</code>	** pow(a, power, modulo=None)
<code>__lshift__</code>	<<
<code>__rshift__</code>	>>
<code>__and__</code>	&
<code>__xor__</code>	^
<code>__or__</code>	

<code>__neg__</code>	-
<code>__pos__</code>	+
<code>__abs__</code>	<code>abs(a)</code>
<code>__invert__</code>	~

Indicizzazione e slicing

L'indicizzazione degli oggetti viene gestita con il metodo speciale `__getitem__()`. L'espressione `obj[index]` viene tradotta nella chiamata al metodo `obj.__getitem__(index)`:

```
>>> class Doubling:
...     def __init__(self, obj):
...         self.obj = obj
...     def __getitem__(self, index):
...         print('Index: ', index)
...         return self.obj[index] * 2
...
>>> a = Doubling('python')
>>> a[1]
Index: 1
'yy'
```

Come:

```
>>> b = Doubling([1, 2, 3, 4])
>>> b[2]
Index: 2
6
>>> b.__getitem__(2)
Index: 2
6
```

Il metodo `__getitem__()` gestisce anche lo slicing:

```
>>> a[:-1]
Index: slice(None, -1, None)
'pythopytho'
>>> a.__getitem__(slice(None, -1, None))
Index: slice(None, -1, None)
'pythopytho'
```

L'indice non deve necessariamente essere un numero intero. Ad esempio, nel caso in cui l'oggetto da indicizzare sia un dizionario, l'indice è in realtà la chiave:

```
>>> d = Doubling({'nome': 'Giovanna', 'cognome': 'Monni'})
>>> d['nome']
Index: nome
'GiovannaGiovanna'
```

Quando una classe definisce il metodo `__getitem__()`, allora le sue istanze sono oggetti iterabili:

```
>>> d = Doubling('abc')
>>> i = iter(d)
>>> type(i)
<class 'iterator'>
```

La chiamata a `next(f)` viene tradotta nella chiamata `d.__getitem__(index)`, con `index` che parte da 0 e viene incrementato di 1 a ogni iterazione, sino a esaurire l'iteratore:

```
>>> next(i)
Index: 0
'aa'
>>> next(i)
Index: 1
'bb'
>>> next(i)
Index: 2
'cc'
>>> next(i)
Index: 3
Traceback (most recent call last):
...
StopIteration
>>> for item in d:
...     print(item)
...
Index: 0
aa
Index: 1
bb
Index: 2
cc
Index: 3
```

Ecco cosa accade se utilizziamo come oggetto iterabile un dizionario:

```
>>> d = Doubling({0: 'zero', 1: 'uno'})
>>> i = iter(d)
>>> next(i)
Index: 0
'zerozero'
>>> next(i)
Index: 1
'unouno'
```

```
>>> next(i)
Index: 2
Traceback (most recent call last):
...
KeyError: 2
```

Le chiamate `next(i)` sono state tradotte nelle chiamate `d.__getitem__(0)`, `d.__getitem__(1)`, `d.__getitem__(2)` e quest'ultima, in corrispondenza di `self.obj[index]`, ovvero di `d[2]`, ha causato un `KeyError`, visto che la chiave 2 non è presente nel dizionario.

NOTA

Per poter utilizzare in modo corretto queste istanze in un contesto di iterazione dovremmo intercettare le eccezioni ed eventualmente lanciarne una di tipo `StopIteration` quando l'iteratore è esaurito.

Quando l'indicizzazione o lo slicing avvengono alla sinistra del simbolo di uguale (si assegnano degli elementi), viene chiamato il metodo `__setitem__()`:

```
>>> class Foo:
...     def __init__(self, obj):
...         self.obj = obj
...     def __setitem__(self, key, value):
...         print("In Foo.__setitem__()")
...         self.obj[key] = value
...     def __repr__(self):
...         return str(self.obj)
...
>>> f = Foo(['a', 'b', 'c'])
>>> f[1] = 'python'
In Foo.__setitem__()
>>> f
['a', 'python', 'c']
>>> f[:-1] = 'aaa', 'bbb'
In Foo.__setitem__()
>>> f
['aaa', 'bbb', 'c']
>>> f = Foo({'a': 1, 'b': 2, 'c': 3})
>>> f['b'] = 'python'
In Foo.__setitem__()
>>> f
{'c': 3, 'b': 'python', 'a': 1}
```

Se, invece, si cancellano degli elementi, viene chiamato il metodo `__delitem__()`:

```

>>> class Foo:
...     def __init__(self, obj):
...         self.obj = obj
...     def __delitem__(self, key):
...         print("In Foo.__delitem__()")
...         del self.obj[key]
...     def __repr__(self):
...         return str(self.obj)
...
>>> f = Foo(['a', 'b', 'c'])
>>> del f[1]
In Foo.__delitem__()
>>> f
['a', 'c']
>>> f = Foo({'a': 1, 'b': 2, 'c': 3})
>>> del f['c']
In Foo.__delitem__()
>>> f
{'b': 2, 'a': 1}

```

Nell'esercizio conclusivo faremo uso sia di `__getitem__()` sia di `__setitem__()`.

Iterazione

Nel Capitolo 1 abbiamo parlato degli iteratori e abbiamo detto che un *iteratore* è un oggetto iterabile che ha i metodi `__iter__()` e `__next__()`. Il metodo `__iter__()` restituisce l'oggetto stesso, mentre `__next__()` restituisce il prossimo elemento dell'iteratore. Possiamo, quindi, creare delle classi che istanziano degli iteratori, semplicemente definendo questi due metodi speciali:

```

>>> class FibonacciIterator:
...     def __init__(self, n):
...         self.n = n
...         self.a, self.b = 1, 2
...     def __iter__(self):
...         return self
...     def __next__(self):
...         if self.a >= self.n:
...             raise StopIteration
...         else:
...             item = self.a
...             self.a, self.b = self.b, self.a + self.b
...         return item

```

```

...
>>> f = FibonacciIterator(6)
>>> next(f)
1
>>> next(f)
2
>>> next(f)
3
>>> next(f)
5
>>> next(f)
Traceback (most recent call last):
...
StopIteration
>>> [i for i in FibonacciIterator(6)]
[1, 2, 3, 5]

```

Quando, invece, definiamo un oggetto iterabile, dobbiamo prestare attenzione al fatto che è bene che due iteratori diversi che iterano sul medesimo oggetto iterabile siano indipendenti, come nel caso seguente:

```

>>> class IterableObject:
...     def __init__(self, value):
...         self.value = value
...     def __iter__(self):
...         return iter(self.value)
...
>>> obj = IterableObject('python')
>>> i1 = iter(obj)
>>> i2 = iter(obj)
>>> next(i1)
'p'
>>> next(i2)
'p'
>>> next(i1)
'y'
>>> next(i1)
't'
>>> next(i2)
'y'

```

Nell'esercizio conclusivo vedremo un altro esempio di oggetto iterabile.

Accesso agli attributi

Nel Capitolo 5 abbiamo visto come i namespace di classe e di istanza siano implementati tramite un dizionario, assegnato all'etichetta `__dict__` della classe o dell'istanza. Abbiamo visto come per default gli attributi vengano cercati prima nel namespace dell'istanza, poi in quello della classe, e infine nei

namespace delle classi base seguendo l'ordine indicato dal MRO.

I metodi `__getattr__()`, `__setattr__()` e `__delattr__()`

Consideriamo il seguente codice:

```
>>> class Foo:
...     def __getattr__(self, name):
...         return 'Attributo %s non disponibile' %name
...
>>> f.foo
'Attributo foo non disponibile'
>>> f.python
'Attributo python non disponibile'
```

Come è facile intuire, il metodo speciale `__getattr__()` consente di definire un comportamento di default nel caso in cui un attributo di istanza non venga trovato seguendo la normale procedura di ricerca:

```
>>> f.python = 'stupendo!'
>>> f.python # Adesso __getattr__() non viene chiamato
'stupendo!'
```

Anche `getattr(instance, name)`, se non trova l'attributo `name`, chiama `instance.__getattr__(name)`:

```
>>> getattr(f, 'foo')
'Attributo foo non disponibile'
```

L'implementazione di `__getattr__()` influisce anche su `hasattr()`. Infatti `hasattr(instance, name)` chiama `getattr(instance, name)` e se quest'ultimo non trova l'attributo seguendo il meccanismo di ricerca di default, allora chiama `__getattr__()`. Se quest'ultimo non solleva una eccezione di tipo `AttributeError`, allora `hasattr()` restituisce sempre `True`, come nel nostro caso:

```
>>> hasattr(f, 'foo')
True
>>> hasattr(f, 'moo')
True
```

Questo significa che, se vogliamo che `hasattr()` si comporti in modo corretto, dobbiamo fare in modo che `__getattr__()` gestisca il comportamento di default per gli attributi di interesse e sollevi una eccezione di tipo `AttributeError` per tutti gli altri:

```
>>> class Foo:
...     def __getattr__(self, name):
...         if name.startswith('py'):
...             return 'python'
...         else:
...             raise AttributeError('Attributo %s non trovato' %name)
...
>>> f = Foo()
>>> f.pyfoo
'python'
>>> f.pymoo
'python'
>>> hasattr(f, 'pyxx')
True
>>> hasattr(f, 'foo')
False
```

La funzione built-in `getattr()`, come sappiamo, prende un terzo argomento opzionale che indica il valore di default da restituire nel caso in cui l'attributo non venga trovato:

```
>>> getattr(f, 'foo')
Traceback (most recent call last):
...
AttributeError: Attributo foo non trovato
>>> getattr(f, 'foo', 'Non trovato!')
'Non trovato!'
```

Il metodo speciale `__getattr__()` viene chiamato anche in questo caso e il valore di default viene restituito solo se `__getattr__()` non gestisce quell'attributo:

```
>>> class Foo:
...     def __getattr__(self, name):
...         print('In __getattr__()')
...         if name.startswith('py'):
...             return 'python'
...         else:
...             raise AttributeError('Attributo %s non trovato' %name)
...
>>> f = Foo()
>>> getattr(f, 'foo', 'Non trovato!')
In __getattr__()
'Non trovato!'
```

Il metodo `__getattr__()` non viene ereditato da `object`, mentre i metodi `__setattr__()` e `__delattr__()` sì:

```
>>> hasattr(object, '__getattr__')
False
>>> hasattr(object, '__setattr__')
True
>>> hasattr(object, '__delattr__')
True
```

Se facciamo l'overriding di `__setattr__()`, ogni assegnamento `instance.name = value` dà luogo alla chiamata `instance.__setattr__(name, value)`:

```
>>> class Foo:
...     def __setattr__(self, name, value):
...         print('In __setattr__()')
...         if name.startswith('py'):
...             super().__setattr__(name, value)
...         else:
...             raise AttributeError('Operazione non consentita')
...
>>> f = Foo()
>>> f.python = 3
In __setattr__()
>>> f.python
3
>>> f.__dict__
{'python': 3}
```

L'assegnamento va fatto usando `super()`. Se proviamo a usare `setattr()`, questa chiamerà, a sua volta, `__setattr__()` e così avremo un loop infinito:

```
>>> class Foo:
...     def __setattr__(self, name, value):
...         if name.startswith('py'):
...             setattr(self, name, value)
...         else:
...             raise AttributeError('Operazione non consentita')
...
>>> f = Foo()
>>> f.python = 3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 4, in __setattr__
...
File "<stdin>", line 4, in __setattr__
RuntimeError: maximum recursion depth exceeded while calling a Python object
```


Infine, se facciamo l'overriding del metodo `__delattr__()`, allora questo viene chiamato quando si usa l'istruzione `del` per cancellare un attributo:

```
>>> class Foo:
...     def __delattr__(self, name):
...         print('In __delattr__()')
...         if name.startswith('py'):
...             super().__delattr__(name)
...         else:
...             raise AttributeError('Operazione non consentita')
...
>>> f = Foo()
>>> f.python = 3
>>> f.__dict__
{'python': 3}
>>> del f.python
In __delattr__()
>>> f.__dict__
{}
>>> del f.foo
In __delattr__()
Traceback (most recent call last):
...
AttributeError: Operazione non consentita
```

Nell'esercizio conclusivo faremo uso sia di `__delattr__()` sia di `__setattr__()`.

L'attributo magico `__getattr__()`

L'attributo `__getattr__()`, a differenza di `__getattr__()`, viene ereditato dalla classe `object`:

```
>>> hasattr(object, '__getattr__')
True
```

Se vogliamo che delle operazioni vengano sempre eseguite quando si accede a un attributo, allora dobbiamo fare l'overriding di `__getattr__()`:

```
>>> class Foo:
...     def __getattr__(self, name):
...         print('In __getattr__()')
...         return super().__getattr__(name)
...
>>> f = Foo()
>>> f.foo
In __getattr__()
```

```
Traceback (most recent call last):
...
AttributeError: 'Foo' object has no attribute 'foo'
```

A differenza di `__getattr__()`, che consente di definire un comportamento di default per gli attributi che non vengono trovati seguendo il look-up standard, `__getattribute__()` viene sempre chiamato, anche se l'attributo esiste:

```
>>> f.foo = 33
>>> f.foo
In __getattribute__()
33
```

Se la classe definisce `__getattr__()`, quest'ultimo, se l'attributo esiste e se `__getattribute__()` non solleva una eccezione di tipo `AttributeError`, non viene chiamato:

```
>>> class Foo:
...     def __getattribute__(self, name):
...         print('In __getattribute__()')
...         return super().__getattribute__(name)
...     def __getattr__(self, name):
...         print('In __getattr__()')
...
>>> f = Foo()
>>> f.foo = 33
>>> f.foo
In __getattribute__()
33
```

Quindi `__getattr__()` viene chiamato solo se l'attributo non esiste:

```
>>> del f.foo
>>> f.foo
In __getattribute__()
In __getattr__()
```

o, più in generale, se `__getattribute__()` solleva una eccezione di tipo `AttributeError`:

```
>>> class Foo:
...     def __getattribute__(self, name):
...         print('In __getattribute__()')
...         if name.startswith('py'):
...             raise AttributeError('Gli attributi py* sono nascosti')
...     else:
```

```

... return super().__getattr__(name)
... def __getattr__(self, name):
...     print('In __getattr__()')
...
>>> f = Foo()
>>> f.foo = 33
>>> f.foo
In __getattr__()
33
>>> f.pyfoo
In __getattr__()
In __getattr__()
>>> f.pyfoo = 33
>>> f.pyfoo
In __getattr__()
In __getattr__()
>>> f.__dict__
In __getattr__()
{'pyfoo': 33, 'foo': 33}

```

Si osservi come `__getattr__()` non venga chiamato se recuperiamo `f.pyfoo` tramite il dizionario:

```

>>> f.__dict__['pyfoo']
In __getattr__()
33

```

Infatti in questo caso viene chiamato `f.__getattr__('__dict__')`, che non inizia con `py`, per cui l'eccezione non viene sollevata. L'istruzione `f.__dict__['pyfoo']` è quindi equivalente alle seguenti:

```

>>> namespace = f.__dict__
In __getattr__()
>>> namespace['pyfoo']
33

```

Descriptors

Un *descriptor* è un oggetto che ha qualcuno tra i metodi `__get__()`, `__set__()` e `__delete__()`. Ad esempio, la seguente classe `MyDescriptor` è un descriptor:

```

>>> class MyDescriptor:
...     def __init__(self, initial_value):
...         self.value = initial_value
...     def __get__(self, instance, instance_type):
...         print('In __get__()')
...         return self.value

```

```
... def __set__(self, instance, value):
...     print('In __set__()')
...     self.value = value
... def __delete__(self, instance):
...     print('In __del__()')
...     del self.value
```

I descriptor vengono utilizzati per definire attributi di classe:

```
>>> class Foo:
...     foo = MyDescriptor(10)
```

Quando accediamo all'attributo in lettura, qualificandolo con la classe o con l'istanza, viene chiamato il metodo `__get__()`:

```
>>> Foo.foo
In __get__()
10
>>> f = Foo()
>>> f.foo
In __get__()
10
```

Quando vi accediamo in scrittura, qualificandolo con l'istanza, viene chiamato il metodo `__set__()`:

```
>>> f.foo = 33
In __set__()
>>> f.foo
In __get__()
33
```

Quando cancelliamo l'attributo, qualificandolo con l'istanza, viene chiamato il metodo `__delete__()`:

```
>>> del f.foo
In __del__()
```

Vediamo di capire meglio cosa viene passato ai metodi del descriptor:

```
>>> class MyDescriptor:
...     def __init__(self, initial_value):
...         self.value = initial_value
```

```

... def __get__(self, instance, instance_type):
...     print('In __get__()', self, instance, instance_type, sep='\n')
...     return self.value
...
... def __set__(self, instance, value):
...     print('In __set__()', self, instance, value, sep='\n')
...     self.value = value
...
... def __delete__(self, instance):
...     print('In __del__()', self, instance, sep='\n')
...     del self.value
...
>>> class Foo:
...     foo = MyDescriptor(10)
...
>>> f = Foo()

```

Al metodo `MyDescriptor.__set__()` vengono passati: come primo argomento, l'istanza del descriptor (in modo implicito), assegnata al parametro `self`; come secondo argomento l'istanza di `Foo`, assegnata al parametro `instance`; come terzo argomento il valore da dare all'attributo, assegnato a `value`:

```

>>> f.foo = 33
In __set__()
<__main__.MyDescriptor object at 0x7f44c362c6d0>
<__main__.Foo object at 0x7f44c362c710>
33

```

Al metodo `MyDescriptor.__get__()` vengono passati: come primo argomento, l'istanza del descriptor (in modo implicito), assegnata al parametro `self`; come secondo argomento l'istanza di `Foo`, assegnata al parametro `instance`; come terzo argomento il tipo di `instance` (`Foo`), assegnato a `instance_type`:

```

>>> f.foo
In __get__()
<__main__.MyDescriptor object at 0x7f44c362c6d0>
<__main__.Foo object at 0x7f44c362c710>
<class '__main__.Foo'>
33

```

Infine, al metodo `MyDescriptor.__delete__()` vengono passati: come primo argomento, l'istanza del descriptor (in modo implicito), assegnata al parametro `self`; come secondo argomento l'istanza di `Foo`, assegnata al parametro `instance`:

```

>>> del f.foo
In __delete__()

```

```
<__main__.MyDescriptor object at 0x7f44c362c210>
<__main__.Foo object at 0x7f44c362c250>
```

Un descriptor che definisce entrambi i metodi `__set__()` e `__get__()`, come nel caso di `MyDescriptor`, viene detto *data descriptor*. I descriptor che definiscono solamente il metodo `__get__()` sono chiamati *non-data descriptor*. Per creare un read-only data descriptor, il metodo `__set__()` deve semplicemente sollevare una eccezione di tipo `AttributeError`.

Lookup degli attributi

Come abbiamo più volte detto, l'accesso agli attributi di un oggetto avviene tramite il suo dizionario `__dict__`. Sinora abbiamo detto che il lookup di `obj.foo` inizia cercando `obj.__dict__['foo']`; se non viene trovato, prosegue con `type(obj).__dict__['foo']` e continua cercando nei dizionari delle classi base, seguendo l'ordine indicato nel MRO di `type(obj)`. Consideriamo, ad esempio, il caso seguente:

```
>>> class A:
...     foo = 'AAA'
...
>>> class B(A):
...     foo = 'BBB'
...
>>> obj = B()
>>> obj.foo = 33
```

Il lookup di `obj.foo` avviene prima controllando se `'foo'` è presente in `obj.__dict__`:

```
>>> 'foo' in obj.__dict__
True
```

Poiché è presente, la ricerca di `obj.foo` termina restituendo `obj.__dict__['foo']`:

```
>>> obj.foo
33
>>> obj.__dict__['foo']
33
```

Se `'foo'` non è presente in `obj.__dict__`, allora si verifica che esso sia presente in `type(obj).__dict__`, ovvero in `B.__dict__`:

```
>>> del obj.foo
>>> obj.__dict__
```

```
{  
>>> 'foo' in B.__dict__  
True
```

Visto che è presente, la ricerca termina restituendo `B.__dict__['foo']`:

```
>>> obj.foo  
'BBB'  
>>> B.__dict__['foo']  
'BBB'
```

Se `'foo'` non è presente in `obj.__dict__` e neppure in `type(obj).__dict__`:

```
>>> del B.foo  
>>> 'foo' in obj.__dict__  
False  
>>> 'foo' in B.__dict__  
False
```

allora viene cercato del dizionario degli attributi delle classi base, seguendo il MRO:

```
>>> type(obj).__mro__  
(<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Viene, quindi, verificato che sia presente in `A.__dict__`, e poiché lo è:

```
>>> 'foo' in A.__dict__  
True
```

la ricerca termina restituendo `A.__dict__['foo']`:

```
>>> obj.foo  
'AAA'  
>>> A.__dict__['foo']  
'AAA'
```

In realtà il meccanismo di ricerca di `obj.foo` è più generico. La modalità di chiamata è diversa a seconda che `obj` sia una classe oppure no. Se `obj` non è una classe, ovvero se `isinstance(obj, type)` restituisce `False`, allora:

- Se `hasattr(type(obj).__dict__['foo'], '__get__')` restituisce `True`, viene effettuata la chiamata al metodo `__get__()`: `type(obj).__dict__['foo'].__get__(obj, type(obj))`;
- Se `hasattr(type(obj).__dict__['foo'], '__get__')` restituisce `False`, viene restituito `obj.__dict__['foo']`.

Se `obj` è una classe, ovvero se `isinstance(obj, type)` restituisce `True`, allora:

- Se `hasattr(obj.__dict__['foo'], '__get__')` restituisce `True`, viene effettuata la chiamata al metodo `__get__()`: `obj.__dict__['foo'].__get__(None, obj)` ;
- Se `hasattr(obj.__dict__['foo'], '__get__')` restituisce `False`, viene restituito `obj.__dict__['foo']`.

Ad esempio, consideriamo il descriptor visto all'inizio della sezione:

```
>>> class MyDescriptor:
...     def __init__(self, initial_value):
...         self.value = initial_value
...     def __get__(self, instance, instance_type):
...         print('In __get__()')
...         return self.value
...     def __set__(self, instance, value):
...         print('In __set__()')
...         self.value = value
...     def __delete__(self, instance):
...         print('In __del__()')
...         del self.value
...
>>> class Foo:
...     foo = MyDescriptor(10)
```

Quando in una espressione compare `Foo.foo`, l'attributo viene trovato in `Foo.__dict__`:

```
>>> Foo.__dict__['foo']
<__main__.MyDescriptor object at 0x7fe5c21cc210>
```

Non viene, però, restituito l'oggetto `<__main__.MyDescriptor object at 0x7fe5c21cc210>`, poiché questo ha l'attributo `__get__`:

```
>>> hasattr(Foo.__dict__['foo'], '__get__')
True
```

Ciò che fa Python, in questo caso, è chiamare il metodo `Foo.__dict__['foo'].__get__()`

```
>>> Foo.__dict__['foo'].__get__(None, Foo)
In __get__()
10
```

I descriptor, quindi, modificano il normale meccanismo di lookup degli

attributi. In precedenza abbiamo visto, infatti, che per default un attributo di istanza nasconde uno di classe con stesso nome:

```
>>> class Foo:
...     foo = 33
...
>>> obj = Foo()
>>> obj.foo
33
>>> obj.foo = 'python'
>>> obj.foo
'python'
>>> Foo.foo
33
```

Nel caso in cui vi sia un descriptor con lo stesso nome dell'attributo di istanza, la regola cambia e il dizionario di classe ha precedenza su quello di istanza, visto che viene eseguita la chiamata al metodo `type(obj).__dict__['foo'].__get__(obj, type(obj))`:

```
>>> class Foo:
...     foo = MyDescriptor(10)
...
>>> obj = Foo()
>>> obj.foo
In __get__()
10
>>> obj.foo = 33
In __set__()
>>> obj.foo
In __get__()
33
```

L'assegnamento `obj.foo = 33` non ha creato un attributo di istanza:

```
>>> obj.__dict__
{}
```

Inoltre, anche se inseriamo esplicitamente un attributo nel dizionario di istanza, questo viene nascosto dal descriptor:

```
>>> obj.__dict__['foo'] = 100
>>> obj.foo
In __get__()
33
>>> obj.__dict__
{'foo': 100}
```

Se l'attributo di classe non è più un descriptor, torniamo al meccanismo di

default:

```
>>> Foo.foo = 'python' # Foo.foo non fa più riferimento ad un descriptor ma a una stringa
>>> obj.foo
100
```

Le property che abbiamo visto nel Capitolo 5 sono implementate con dei descriptor. Facciamo un po' di memoria locale e riprendiamo quell'esempio. Partendo dal codice seguente:

```
>>> import math
>>> class Circle:
...     def __init__(self, r=1):
...         self.radius = r
...     def area(self):
...         return math.pi * self.radius**2
...
>>> c = Circle()
>>> c.area()
3.141592653589793
```

avevamo poi decorato il metodo `Circle.area()` con la classe built-in `property`, in modo da ottenere l'area usando la notazione `c.area` piuttosto che `c.area()`:

```
>>> class Circle:
...     def __init__(self, r=1):
...         self.radius = r
...     @property
...     def area(self):
...         return math.pi * self.radius**2
...
>>> c = Circle()
>>> c.area # È come se eseguiamo `c.area()`
3.141592653589793
```

Come possiamo vedere, il seguente esempio mostra la parentela tra `property` e `descriptor`:

```
>>> import math
>>> class Area:
...     def __get__(self, obj, obj_type):
...         return math.pi * obj.radius**2
...
>>> class Circle:
```

```
... def __init__(self, r=1):  
...     self.radius = r  
...     area = Area()  
...  
>>> c = Circle()  
>>> c.area  
3.141592653589793
```

Anche la classe `super`, i metodi statici e quelli di classe sono implementati tramite dei descriptor. Per maggiori informazioni si può consultare la pagina <http://docs.python.org/3/howto/descriptor.html>.

Metaclassi

Il termine metaclassa tipicamente viene associato a qualcosa di astruso e difficile. In realtà è un concetto molto semplice: una *metaclassa* è una classe che, quando viene chiamata, ha l'effetto di creare e restituire una nuova classe. Ad esempio, data una classe `Foo`, se effettuiamo la chiamata `Foo()` e questa genera come istanza una classe, allora `Foo` è una metaclassa.

NOTA

In realtà una metaclassa non è necessariamente un tipo, ovvero una istanza di `type`, ma, più in generale, è un qualsiasi oggetto callable che accetta gli stessi argomenti di `type`, che crea una classe e la restituisce.

Se riflettiamo un attimo, ci rendiamo conto che per tutto il corso del libro abbiamo fatto uso di una metaclassa senza saperlo: la classe `type`. Quando, infatti, effettuiamo una chiamata a `type`, passandole un oggetto, questa restituisce il tipo (la classe) dell'oggetto. Ad esempio, se passiamo a `type` una lista, allora restituisce la classe `list`:

```
>>> instance = type([1, 2, 3]) # Crea una istanza
>>> instance # L'istanza creata è la classe 'list'
<class 'list'>
```

L'etichetta `instance` e l'etichetta `list` fanno riferimento alla medesima classe:

```
>>> instance is list
True
>>> instance('python') # Equivalente a list('python')
['p', 'y', 't', 'h', 'o', 'n']
```

Il mito della magia più profonda

Il 17 dicembre 2002, Michele Simionato pubblicò su **comp.lang.python** un messaggio dal titolo *Python as an Object Oriented Programming Language*, dove pose alcune domande sulle classi di nuovo stile introdotte con Python 2.2 e, più in generale, sulla programmazione a oggetti. Si iniziò a discutere su questi temi e sulle metaclassi e qualcuno manifestò alcune perplessità in

merito alle nuove funzionalità introdotte:

Non voglio dire che molte delle aggiunte al linguaggio non siano benvenute, ma solo che sembra che riguardino argomenti e pratiche di programmazione complicati.

Il 23 dicembre 2002, in risposta a questo messaggio, Tim Peters scrisse la ormai famosa frase:

[Le metaclassi] sono la magia più profonda, per cui il 99% degli utenti di Python non dovrebbe mai preoccuparsi di esse. Se ti chiedi se ti possono servire, allora non ti servono (le persone che realmente necessitano delle metaclassi sanno con certezza che le devono usare e non hanno bisogno di alcuna spiegazione sul motivo per cui devono usarle).

Questa frase viene usualmente riportata quando si vuole introdurre il concetto di metaclasse e questo è uno dei motivi per cui si pensa che vi sia un alone di mistero attorno a esse. Il nostro scopo è sfatare questo mito. Come vedremo, non vi è niente di misterioso o di complicato dietro il concetto di metaclasse. Per quanto ci riguarda, sappiamo già di cosa si tratta, per cui non ci resta che studiare più in dettaglio la metaclasse `type`, vedere come creare nuove metaclassi e analizzare dei casi pratici di utilizzo.

La metaclasse `type`

Nella sezione *Il modello a oggetti di Python* abbiamo detto che tutte le classi sono istanze della classe `type`, compresa `type` stessa:

```
>>> instance = type(type)
>>> instance
<class 'type'>
>>> instance is type
True
```

NOTA

L'unica metaclasse built-in è la classe `type`. La classe `object` non è una metaclasse. Le sue istanze non sono classi, e infatti non sono istanze di `type`:

```
>>> isinstance(object(), type)
False
```

Oltre a restituire il tipo (la classe) di un oggetto, la classe `type` ci consente di creare delle nuove classi. Per fare ciò dobbiamo passarle tre argomenti: il nome che vogliamo dare alla classe, una tupla di classi base e un dizionario degli attributi. Ad esempio, questa definizione di `Foo`:

```
>>> class Foo:  
... pass
```

è equivalente alla seguente:

```
>>> Foo = type('Foo', (), {})
```

Il nome della classe non è legato a quello dell'etichetta:

```
>>> Foo = type('MyClass', (), {})  
>>> Foo.__name__  
'MyClass'
```

Il dizionario passato come terzo argomento consente di specificare gli attributi della classe. Questa definizione:

```
>>> class Foo:  
... foo = 100
```

è equivalente alla seguente:

```
>>> Foo = type('Foo', (), {'foo': 100})  
>>> Foo.foo  
100
```

Ovviamente possiamo anche creare delle classi che hanno dei metodi. Questa definizione:

```
>>> class Foo:  
... def __init__(self, value):  
...     self.value = value
```

è equivalente alla seguente:

```
>>> def __init__(self, value):  
...     self.value = value  
...
```

```
>>> Foo = type('Foo', (), {'__init__': __init__})
```

infatti:

```
>>> f = Foo(10)
>>> f.value
10
```

Il secondo argomento di `type` ci consente di stabilire la gerarchia di ereditarietà. Data una classe `A`:

```
>>> class A:
...     a = 'aaa'
```

la seguente definizione di `B`:

```
>>> class B(A):
...     pass
```

è equivalente a questa:

```
>>> B = type('B', (A,), {})
```

La classe `B` è infatti una sottoclasse di `A`:

```
>>> issubclass(B, A)
True
>>> B.a
'aaa'
```

Non è necessario inserire `object` nella tupla delle classi base e l'ordine delle classi nel MRO è stabilito sulla base dell'ordine dato alle classi nel secondo argomento di `type`:

```
>>> class A:
...     pass
...
>>> class B:
...     pass
...
>>> class C:
...     pass
...
>>> Foo = type('Foo', (B, A, C), {})
```

```
>>> Foo.__mro__  
(<class '__main__.Foo'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.C'>, <class 'object'>)
```

Ora che conosciamo meglio la metaclassa `type`, vediamo come possiamo definire nuove metaclassi.

Personalizzare il processo di creazione delle classi

Come abbiamo detto più volte, la classe `type` è per default la metaclassa di tutte le classi, nel senso che ogni classe, se non specificato diversamente, viene istanziata da `type`. Quindi, per default, data una classe `Foo`, come ben sappiamo, `type(Foo)` restituisce `type`:

```
>>> class Foo:
...     pass
...
>>> type(Foo) is type
True
```

Se vogliamo possiamo modificare questo comportamento di default, creando una nuova metaclassa che istanzi le nostre classi.

Creare nuove metaclassi

Se vogliamo definire una metaclassa, dobbiamo semplicemente creare una classe che eredita da una metaclassa. Ci serve, quindi, una metaclassa da cui ereditare; al momento abbiamo a disposizione solamente `type`, che è l'unica metaclassa built-in. Quindi, per creare una metaclassa chiamata `FooMeta`, dobbiamo definire una classe `FooMeta` che erediti dalla metaclassa `type`:

```
>>> class FooMeta(type):
...     pass
```

Niente di più semplice: la classe `FooMeta` eredita da una metaclassa, per cui è anch'essa una metaclassa. Possiamo quindi dire che una classe `MyClass` è una metaclassa se e solo se è sottoclasse di `type`, ovvero se e solo se `issubclass(MyMeta, type)` restituisce `True`, come nel caso di `FooMeta`:

```
>>> issubclass(FooMeta, type)
True
```

Non abbiamo definito alcun attributo di `FooMeta`, per cui questa eredita tutto da `type` e si comporta esattamente come quest'ultima. Se le passiamo un oggetto, restituisce il tipo di quest'ultimo:

```
>>> FooMeta([])
<class 'list'>
```

Se le passiamo tre argomenti, rispettivamente il nome della classe da istanziare, le sue classi base e il dizionario degli attributi, allora `FooMeta` crea una nuova classe, esattamente come fa `type`:

```
>>> Foo = FooMeta('Foo', (), {'foo': 100})
>>> Foo.foo
100
```

La cosa importante è che cambia il comportamento di default, perché `Foo` viene istanziata direttamente da `FooMeta` e non da `type`:

```
>>> type(Foo)
<class '__main__.FooMeta'>
```

Questo è importante perché significa che possiamo definire dei metodi per `FooMeta` e se, ad esempio, definiamo `FooMeta.__new__()` e `FooMeta.__init__()`, allora possiamo personalizzare la fase di creazione e inizializzazione delle classi. Vedremo tra breve come fare, ma prima di andare avanti riassumiamo brevemente quanto abbiamo visto in questa sezione. Abbiamo definito una metaclassa `FooMeta`, ovvero una classe che eredita da una metaclassa:

```
>>> class FooMeta(type):
...     pass
```

Le istanze della metaclassa `FooMeta` sono, a loro volta, delle classi, e sono oggetti di tipo `FooMeta` e non di tipo `type`:

```
>>> Foo = FooMeta('Foo', (), {})
>>> type(Foo)
<class '__main__.FooMeta'>
```

La classe `FooMeta` viene istanziata dalla metaclassa `type`, per cui è un oggetto di tipo `type`:

```
>>> type(FooMeta)
<class 'type'>
```

La classe `type` è metaclassa di se stessa:

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

Come abbiamo più volte detto in questa sezione, affinché una classe sia metaclassa, è sufficiente che erediti da una metaclassa qualsiasi e non necessariamente in modo diretto da `type`:

```
>>> class MooMeta(FooMeta):
...     pass
...
>>> MooMeta('Moo', (), {}) # Le istanze di MooMeta sono delle classi
<class '__main__.Moo'>
```

Questo è ovvio, in base a quanto detto nella sezione *L'ereditarietà* del [Capitolo 5](#), quando abbiamo parlato della relazione *is-a*.

Vediamo ora come possiamo definire una classe, utilizzando l'istruzione `class`, specificando che la sua metaclassa non deve essere `type` ma un'altra metaclassa.

Scegliere la metaclassa delle nostre classi

Come sappiamo, quando creiamo una classe possiamo passarle degli argomenti che rappresentano le sue classi base. Ad esempio, nel caso seguente la classe `Foo` eredita da `A`:

```
>>> class A:
...     pass
...
>>> class Foo(A):
...     pass
```

Possiamo passare, oltre all'elenco delle classi base, anche un argomento keyword-only chiamato `metaclass`, che serve per specificare la metaclassa da utilizzare per istanziare `Foo`. Per default si ha `metaclass=type`, per cui la seguente definizione della classe `Foo`:

```
>>> class Foo:
...     pass
...
>>> type(Foo)
<class 'type'>
```

è equivalente a questa:

```
>>> class Foo(metaclass=type):
...     pass
...
```

```
>>> type(Foo)
<class 'type'>
```

Questo è il motivo per cui tutte le classi per default vengono istanziate da `type`. Possiamo, quindi, modificare questo comportamento di default, passando esplicitamente la metaclass da utilizzare:

```
>>> class FooMeta(type):
...     pass
...
>>> class Foo(metaclass=FooMeta):
...     pass
...
>>> type(Foo)
<class '__main__.FooMeta'>
```

Ancora una volta, come possiamo vedere, tutto è semplice e lineare. Non c'è niente di complesso e astruso.

NOTA

In Python 2 la metaclass non va specificata tramite l'assegnamento all'argomento `metaclass`, ma attraverso l'assegnamento all'attributo `__metaclass__`. Quindi, se in Python 3 scriviamo:

```
class Foo(metaclass=FooMeta):
    pass
```

in Python 2 dobbiamo scrivere:

```
class Foo(object):
    __metaclass__ = FooMeta
```

Più avanti, nella sezione *Metaclassi che accettano argomenti opzionali*, vedremo come nell'istruzione `class` possiamo passare anche altri argomenti oltre a `metaclass`.

Concludiamo questa sezione osservando che se una classe `A` ha metaclass `AMeta`, allora se `B` eredita da `A`, giustamente ha anch'essa `AMeta` come metaclass:

```
>>> class AMeta(type):
...     pass
...
```

```
>>> class A(metaclass=AMeta):
...     pass
...
>>> class B(A):
...     pass
...
>>> type(B)
<class '__main__.AMeta'>
```

La creazione delle classi

Sappiamo che le classi vengono istanziate dalle metaclassi, ma ancora non sappiamo in quale momento la metaclassa crei la classe e in cosa consista questo processo di creazione. Tutto ciò è argomento di questa sezione e, come vedremo, ancora una volta non vi è nulla di complicato.

Detto questo, vediamo di capire cosa accade al termine dell'istruzione `class`. Innanzitutto ricordiamoci che, mentre la suite di una istruzione `def` viene eseguita solamente quando la funzione viene chiamata:

```
>>> def foo():
...     print('In foo()')
...
>>> foo()
In foo()
```

la suite di una istruzione `class` viene eseguita immediatamente:

```
>>> class Foo:
...     print('In Foo')
...     a = 33
...
In Foo
```

Come possiamo vedere, è stato mostrato l'output della `print()`, a conferma del fatto che la suite è stata immediatamente eseguita. Un'altra prova è l'esistenza del dizionario degli attributi:

```
>>> Foo.__dict__['a']
33
```

La suite della clausola `class` viene eseguita immediatamente perché è necessario creare il dizionario degli attributi, stabilire quali sono le eventuali classi base e, soprattutto, indicare la metaclassa. Ad esempio, nel seguente caso:

```
>>> class Foo(list):
...     """Una classe inutile."""
...     a = 33
```

vengono estrapolate dall'istruzione `class` le seguenti informazioni:

1. la *metaclass*, sulla base dell'assegnamento fatto all'argomento keyword-only `metaclass`, che in questo caso assume il suo valore di default `metaclass=type`;
2. il *nome* della classe, che in questo caso è la stringa `'Foo'`;
3. le *classi base* indicate in modo esplicito (`object` è implicita): in questo caso è la sola classe `list`;
4. il *dizionario degli attributi*, contenente sia quelli indicati esplicitamente, come ad esempio `a`, sia quelli creati implicitamente, come ad esempio `__doc__`.

A chi servono queste informazioni? La risposta è scontata quanto significativa: servono all'interprete per creare la classe. Infatti la precedente istruzione `class` (clausola più suite) viene tradotta da Python in questo assegnamento:

```
Foo = type('Foo', (list,), {'a': 33, '__doc__': 'Una classe inutile.'})
```

La classe viene quindi creata e assegnata con l'istruzione precedente. Anche se corriamo il rischio di essere ripetitivi, proviamo a chiarire ancora meglio come avviene questo assegnamento:

- l'etichetta alla sinistra del simbolo di assegnamento è `Foo` perché questa è l'etichetta che abbiamo usato nella clausola `class` (`class Foo(list):`);
- viene chiamata la metaclass `type` perché è quella indicata (valore di default) nella clausola `class` (`class Foo(list, metaclass=type):`);
- vengono passati alla metaclass `type` tre argomenti:
 1. il nome da dare alla classe, ricavato sulla base dell'etichetta usata nella clausola `class`: `'Foo'`;
 2. una tupla contenente le classi base indicate nella clausola `class`: `(list,)`;
 3. il dizionario degli attributi della classe: `{'a': 33, '__doc__': 'Una classe inutile.'}`.

Facciamo un altro passo e chiediamoci cosa comporta realmente la precedente chiamata alla metaclass `type`. Partiamo da lontano e riprendiamo quanto abbiamo visto nella sezione *Il metodo speciale `__call__()`* di questo capitolo. Avevamo detto che se una classe `Foo` definisce il metodo speciale `__call__()`, allora le sue istanze sono oggetti callable e la loro chiamata viene tradotta nella

chiamata al metodo `Foo.__call__()`. Per essere più precisi, avevamo detto che se `f` è una istanza di `Foo`, allora la chiamata `f()` viene tradotta in `type(f).__call__(f)`:

```
>>> class Foo:
...     def __call__(self):
...         print('In Foo.__call__()')
...
>>> f = Foo()
>>> f() # Viene tradotta in type(f).__call__(f), ovvero Foo.__call__(f)
In Foo.__call__()
```

Questo significa che la chiamata `type('Foo', (list,), {'a': 33, '__doc__': 'Una classe inutile.'})` viene tradotta in `type(type).__call__(type, 'Foo', (list,), {'a': 33, '__doc__': 'Una classe inutile.'})`. Questo è la chiave di tutto e, una volta capito questo, il mito della magia più profonda è sfatato. Per essere certi di averlo sfatato definitivamente, vediamo ancora un breve esempio, che possiamo considerare come il riassunto di questa sezione:

```
>>> class FooMeta(type):
...     def __call__(meta, name, bases, namespace):
...         print('FooMeta.__call__()')
...
>>> class InstanceOfFooMeta(type, metaclass=FooMeta):
...     pass
...
>>> class Foo(metaclass=InstanceOfFooMeta):
...     pass
...
FooMeta.__call__()
```

Il risultato di questo esempio è la chiamata al metodo `FooMeta.__call__()`. Vediamo di capire cosa è successo, esaminando una per una le varie istruzioni `class`, partendo dalla classe `FooMeta`. Qui viene passato all'argomento `metaclass` il valore di default `metaclass=type`, per cui al termine della suite, in base a quanto abbiamo detto in precedenza, l'istruzione `class` viene tradotta in:

```
FooMeta = type('FooMeta', (type,), {'__call__': ...})
```

Per semplicità abbiamo indicato con tre punti i restanti elementi del dizionario degli attributi. Questa chiamata, come sappiamo, viene tradotta in:

```
FooMeta = type(type).__call__(type, 'FooMeta', (type,), {'__call__': ...})
```

ovvero:

```
FooMeta = type.__call__(type, 'FooMeta', (type,), {'__call__': ...})
```

A questo punto è stata creata la classe, la quale è stata assegnata all'etichetta `FooMeta`. Questa è una metaclassa, perché eredita da `type`.

Nella seconda istruzione `class` viene passata al parametro `metaclass` la metaclassa `FooMeta`, per cui la classe `InstanceOfFooMeta` viene creata dalla metaclassa `FooMeta`. La creazione di `InstanceOfFooMeta` avviene quindi con la seguente istruzione di assegnamento, eseguita al termine dell'istruzione `class`:

```
InstanceOfFooMeta = FooMeta('InstanceOfFooMeta', (type,), {...})
```

Questa viene tradotta in:

```
InstanceOfFooMeta = type(FooMeta).__call__(FooMeta, 'InstanceOfFooMeta', (type,), {...})
```

Poiché `type(FooMeta)` restituisce `type`, si ha:

```
InstanceOfFooMeta = type.__call__(FooMeta, 'InstanceOfFooMeta', (type,), {...})
```

A questo punto è stata creata la classe, la quale è stata assegnata all'etichetta `InstanceOfFooMeta`. Anche questa è una metaclassa, perché eredita da `type`.

Nell'ultima istruzione `class` viene passata al parametro `metaclass` la metaclassa `InstanceOfFooMeta`, per cui la classe `Foo` viene creata dalla metaclassa `InstanceOfFooMeta`. La creazione di `Foo` avviene, quindi, con la seguente istruzione di assegnamento, eseguita al termine dell'istruzione `class`:

```
Foo = InstanceOfFooMeta('Foo', (), {...})
```

Questa viene tradotta in:

```
Foo = type(InstanceOfFooMeta).__call__(InstanceOfFooMeta, 'Foo', (), {...})
```

Poiché `InstanceOfFooMeta` è stata istanziata da `FooMeta`, `type(InstanceOfFooMeta)` restituisce `FooMeta`, per cui in definitiva si ha:

```
Foo = FooMeta.__call__(InstanceOfFooMeta, 'Foo', (), {...})
```

Quest'ultima stampa la stringa `'FooMeta.__call__()'` e poi restituisce `None`, per cui in definitiva l'etichetta `Foo` fa riferimento all'oggetto `None`:


```
>>> Foo is None
True
```

Ovviamente il risultato finale ha poco senso, ma non ci interessa, perché queste righe di codice ci hanno consentito di illustrare nel dettaglio il meccanismo che sta alla base della creazione delle classi. Vedremo degli esempi più concreti a mano a mano che andremo avanti con la trattazione.

La preparazione del dizionario degli attributi

Abbiamo appena visto come al termine dell'istruzione `class` la metaclassa crei la classe e, per fare ciò, viene chiamata passandole il nome della classe, le classi base e il dizionario degli attributi. Sull'origine dei primi due argomenti non abbiamo niente da aggiungere rispetto a quanto visto nella sezione precedente: entrambi sono esplicitamente indicati nella clausola `class`. Per quanto riguarda il dizionario degli attributi, il discorso è diverso. Non tutti gli elementi del dizionario sono esplicitamente indicati nell'istruzione `class` (clausola più suite):

```
>>> class FooMeta(type):
...     def __new__(metaccls, name, bases, namespace):
...         print(namespace)
...
>>> class Foo(metaclass=FooMeta):
...     """Documentazione..."""
...     a = 33
...
...     {'__doc__': 'Documentazione...', '__qualname__': 'Foo', '__module__': '__main__', 'a': 33}
```

Infatti vengono implicitamente creati alcuni attributi speciali, come `__qualname__`, `__module__` e `__doc__`. Non solo, vengono implicitamente fatte anche alcune trasformazioni:

```
>>> class Foo(metaclass=FooMeta):
...     __a = 33
...
...     {'_Foo__a': 33, '__qualname__': 'Foo', '__module__': '__main__'}
```

Come possiamo osservare, l'attributo di classe `__a` viene trasformato in `__Foo__a`. Questo perché gli attributi di classe che iniziano, ma non finiscono, con due underscore, come abbiamo visto nel [Capitolo 5](#), vengono automaticamente trasformati facendoli precedere da un underscore e dal nome della classe, in modo da renderli nascosti.

Il dizionario deve essere già pronto prima della chiamata alla metaclassa, visto che è proprio un argomento di quest'ultima. Per intenderci, nel nostro esempio l'istruzione `class` viene tradotta nella chiamata:

```
Foo = FooMeta('Foo', (), {'__qualname__': 'Foo', ...})
```

quindi il dizionario degli attributi deve essere immediatamente preparato al termine dell'istruzione `class`, come prima cosa da fare. La domanda nasce spontanea: chi prepara il dizionario? Il metodo speciale `__prepare__()` della metaclassa. La chiamata a questo metodo è quindi il primo passo che viene compiuto durante il processo di creazione della classe. Se la metaclassa non lo definisce, come sappiamo, esso viene cercato seguendo il suo MRO. Ad esempio, se la metaclassa non definisce `__prepare__()` ed eredita solamente da `type`, allora viene chiamato `type.__prepare__()`. Quest'ultimo, come possiamo vedere, prende un numero arbitrario di argomenti e restituisce un dizionario:

```
>>> type.__prepare__(1, 2, 3, a=11, b=22, c=33)
{}

```

NOTA

Il dizionario restituito da `__prepare__()` può essere un qualsiasi oggetto mappatura e non necessariamente un dizionario ordinario.

Detto ciò, possiamo scendere ancor più nel dettaglio nell'analisi del processo di creazione delle classi. I passi che vengono compiuti per creare la classe sono, nell'ordine, i seguenti:

1. viene chiamato il metodo `__prepare__()` della metaclassa, il quale crea e restituisce il dizionario degli attributi;
2. Python popola il dizionario restituito da `__prepare__()`, aggiungendo gli attributi speciali (`__module__`, `__qualname__` ecc.); fatto ciò, trasforma gli attributi che dovranno essere nascosti (nel precedente esempio, `__a` viene trasformato in `__Foo__a`) e aggiunge anche questi al dizionario;
3. viene eseguita la chiamata alla metaclassa e viene assegnato il risultato all'etichetta (nel precedente esempio: `Foo = FooMeta('Foo', (), {'__qualname__': 'Foo', ...})`).

Fermiamoci a riflettere sul metodo `__prepare__()`, in modo da non farci sfuggire alcuni importanti dettagli. Innanzitutto, vediamo come avviene la chiamata a

questo metodo.

La chiamata al metodo `__prepare__()`

Nel momento in cui viene chiamato il metodo `__prepare__()`, l'istanza della metaclassa (la classe che vogliamo creare) non è stata ancora creata, perché la metaclassa non è stata ancora chiamata. Quindi la chiamata a `__prepare__()` non può avvenire qualificando il metodo tramite l'istanza, ma è necessario farlo tramite la classe (nel nostro caso, ad esempio, viene effettuata la chiamata `FooMeta.__prepare__()` e non `Foo.__prepare__()`, perché la classe `Foo` non è stata ancora creata). Sulla base di quanto abbiamo detto nella sezione *I metodi e le property* del Capitolo 5, se un metodo viene qualificato tramite la classe e non viene reso esplicitamente di classe decorandolo con `classmethod`, allora è un metodo statico e non gli viene passato implicitamente alcun argomento.

Nel nostro caso, la classe `FooMeta` non fa l'overriding del metodo `type.__prepare__()`, per cui viene chiamato quest'ultimo. Nonostante possa prendere un numero arbitrario di argomenti, gli vengono passati esplicitamente due argomenti: il nome della classe e la tupla delle classi base. Questo significa che, se facciamo l'overriding di `__prepare__()`, dobbiamo necessariamente indicare che prende due argomenti:

```
>>> class FooMeta(type):
...     def __prepare__(clsname, bases):
...         print(clsname, bases, sep='\n')
...         return {}
...
>>> class Foo(metaclass=FooMeta):
...     pass
...
Foo
()
```

Quindi, se ne specifichiamo un numero diverso da due, otteniamo un errore:

```
class FooMeta(type):
...     def __prepare__(clsname):
...         return {}
...
>>> class Foo(metaclass=FooMeta):
...     pass
...
Traceback (most recent call last):
...
```

TypeError: __prepare__() takes 1 positional argument but 2 were given

Se ci interessa avere un riferimento alla metaclassa, dobbiamo rendere `__prepare__()` metodo di classe. In questo caso, oltre ai due argomenti espliciti, dobbiamo ovviamente dichiarare anche il primo, passato implicitamente:

```
>>> class FooMeta(type):
...     @classmethod
...     def __prepare__(metacls, clsname, bases):
...         print(metacls, clsname, bases, sep='\n')
...         return {}
...
>>> class Foo(metaclass=FooMeta):
...     pass
...
<class '__main__.FooMeta'>
Foo
()
```

NOTA

Quando si crea una classe chiamando direttamente la metaclassa, il metodo `__prepare__()` non viene chiamato:

```
>>> class FooMeta(type):
...     def __prepare__(clsname, bases):
...         print('In FooMeta.__prepare__()')
...
>>> class FooMeta(type):
...     def __prepare__(clsname, bases):
...         print('In FooMeta.__prepare__()')
...         return {'fooattr': 'foo'}
...
>>> class Foo(metaclass=FooMeta):
...     pass
...
In FooMeta.__prepare__()
>>> hasattr(Foo, 'fooattr')
True
>>> Foo = FooMeta('Foo', (), {}) # __prepare__() non viene chiamato
>>> hasattr(Foo, 'fooattr')
False
```

A partire da Python 3.3, il modulo `types` definisce la funzione `types.prepare_class()` che, come suggerisce il nome, prepara la classe eseguendo il metodo

`__prepare__()`. Questa prende il nome della classe, la tupla delle classi base e gli argomenti da passare alla classe, e restituisce una tupla contenente la metaclass, il dizionario restituito da `__prepare__()` e gli argomenti `kwds`:

```
>>> from types import prepare_class as pc
>>> metaclass, ns, kwds = pc('Foo', (), kwds={'metaclass': FooMeta})
In FooMeta.__prepare__()
>>> ns
{'fooattr': 'foo'}
>>> Foo = metaclass('Foo', (), ns)
>>> Foo.fooattr
'foo'
```

Adesso vediamo come viene popolato il dizionario restituito da `__prepare__()` e come viene creato il namespace vero e proprio della classe.

Creazione dell'attributo `__dict__` della classe

Dopo che il metodo `__prepare__()` restituisce il dizionario, l'interprete aggiunge gli attributi speciali, trasforma quelli che devono essere nascosti e aggiunge anche questi. Per evidenziare ciò, possiamo stampare il dizionario passato al costruttore. Infatti gli attributi non inseriti con il metodo `__prepare__()` ma presenti nel dizionario passato al costruttore sono quelli aggiunti implicitamente da Python:

```
>>> class FooMeta(type):
...     def __prepare__(clsname, bases):
...         return {'x': 0}
...     def __new__(metacls, name, bases, namespace):
...         print(namespace)
...         return super().__new__(metacls, name, bases, namespace)
...
>>> class Foo(metaclass=FooMeta):
...     """Una classe inutile."""
...     __a = 1
...
{'__qualname__': 'Foo', 'x': 0, '__module__': '__main__', '_Foo__a': 1, '__doc__': 'Una classe inutile.'}
```

Quindi `FooMeta.__prepare__()` ha restituito il dizionario `{'x': 0}` e tutti gli altri elementi sono stati aggiunti automaticamente da Python.

Il dizionario passato al metodo `FooMeta.__new__()` non è l'oggetto a cui fa riferimento `Foo.__dict__`. Questo perché l'attributo `Foo.__dict__` è creato e assegnato nel metodo `type.__new__()`, il quale prende come argomento il dizionario restituito

da `FooMeta.__prepare__()` e crea, a partire da questo, un nuovo oggetto proxy:

```
>>> class FooMeta(type):
...     def __prepare__(clsname, bases):
...         return {'foo': 100}
...     def __new__(metacls, name, bases, namespace):
...         cls = super().__new__(metacls, name, bases, namespace)
...         print("Tipo dell'argomento `namespace`:", type(namespace))
...         print("Tipo di cls.__dict__:", type(cls.__dict__))
...         return cls
...
>>> class Foo(metaclass=FooMeta):
...     """Una classe inutile."""
...     pass
...
Tipo dell'argomento `namespace`: <class 'dict'>
Tipo di cls.__dict__: <class 'mappingproxy'>
```

Come abbiamo visto nel [Capitolo 5](#), infatti, in Python 3 il dizionario degli attributi di classe non è un oggetto di tipo `dict`, ma di tipo `types.MappingProxyType`:

```
>>> import types
>>> type(Foo.__dict__) is types.MappingProxyType
True
```

Vediamo adesso alcuni esempi più completi di utilizzo delle metaclassi.

Alcuni esempi di utilizzo delle metaclassi

Sinora nei nostri esempi abbiamo sempre definito delle metaclassi piuttosto banali, ma non inutili, poiché ci hanno consentito di capire la sostanza con degli esempi molto semplici. Possiamo quindi fare un ulteriore passo avanti, e vedere qualche altro esempio, partendo da un caso molto istruttivo sull'overwriting del metodo `__prepare__()`.

Modificare il prefisso degli attributi nascosti

Sia nel [Capitolo 5](#), sia nelle sezioni precedenti, abbiamo visto come gli attributi di classe che iniziano, ma non finiscono, con due underscore, vengano automaticamente nascosti da Python. Ad esempio, se una classe `Foo` definisce un attributo `__a`, questo viene automaticamente trasformato in `_Foo__a`:

```
>>> class Foo:
```

```
... __a = 1
...
>>> hasattr(Foo, '__a'), hasattr(Foo, '_Foo__a')
(False, True)
>>> Foo.__dict__['_Foo__a']
1
```

Su questa trasformazione non possiamo intervenire, perché avviene dopo che è stato chiamato il metodo `__prepare__` e prima che venga chiamato il metodo `__new__()`. Possiamo, però, creare in `__prepare__()` un oggetto di tipo `dict` che intercetta l'assegnamento e cambia il prefisso di questi attributi nascosti. Andiamo, quindi, per ordine, e definiamo un dizionario che intercetta l'assegnamento:

```
>>> class Namespace(dict):
...     def __setitem__(self, name, value):
...         print('In __setitem__:', name, value)
...         super().__setitem__(name, value)
...
>>> n = Namespace()
>>> n
{}
```

L'aggiunta a `n` di una nuova coppia chiave-valore viene intercettata da `n.__setitem__()`:

```
>>> n['python'] = 3
In __setitem__: python 3
>>> n
{'python': 3}
```

Se ora in `FooMeta.__prepare__()` restituiamo un oggetto di tipo `Namespace`, questo ci consente di osservare sia che gli attributi speciali vengono aggiunti al dizionario prima della chiamata a `FooMeta.__new__()`, sia che la trasformazione da `__a` a `_Foo__a` avviene anch'essa prima della chiamata a `FooMeta.__new__()`:

```
>>> class FooMeta(type):
...     def __prepare__(clsname, bases):
...         return Namespace()
...     def __new__(metacls, name, bases, namespace):
...         print('In __new__()')
...         return super().__new__(metacls, name, bases, namespace)
...
>>> class Foo(metaclass=FooMeta):
```

```

... __a = 33
...
In __setitem__: __module__ __main__
In __setitem__: __qualname__ Foo
In __setitem__: _Foo__a 33
In __new__()

```

Detto ciò, se vogliamo cambiare il prefisso degli attributi nascosti, possiamo farlo nel modo seguente:

```

>>> class Namespace(dict):
...     def __init__(self, clsname):
...         self.clsname = clsname
...         super().__init__()
...     def __setitem__(self, name, value):
...         prefix = '_' + self.clsname
...         name = name if not name.startswith(prefix) else name.replace(prefix, '_hidden')
...         super().__setitem__(name, value)
...
>>> class ChangeHiddenMeta(type):
...     def __prepare__(clsname, bases):
...         return Namespace(clsname)
...
>>> class Foo(metaclass=ChangeHiddenMeta):
...     __a = 33

```

Come possiamo vedere, rispetto alla classe `Namespace` precedentemente definita, è stato aggiunto il metodo `Namespace.__init__()`, in modo da memorizzare il nome della classe, poiché questo serve poi al metodo `Namespace.__setitem__()`. Quest'ultimo verifica se l'attributo è nascosto, nel qual caso sostituisce il prefisso con un altro (nella fattispecie, per la classe `Foo` sostituisce il prefisso `_Foo` con `_hidden`):

```

>>> hasattr(Foo, '__a'), hasattr(Foo, '_Foo__a'), hasattr(Foo, '_hidden__a')
(False, False, True)
>>> Foo._hidden__a
33

```

In definitiva, abbiamo definito una metaclassa che modifica il prefisso di default degli attributi nascosti. Visto che la trasformazione viene realizzata dalla metaclassa `ChangeHiddenMeta`, la trasformazione è trasparente all'utente che definisce delle classi aventi come metaclassa `ChangeHiddenMeta`. L'esercizio visto in questa sezione è un altro buon esempio che ci consente di

fare ulteriore pratica con le metaclassi, ma non lo si deve utilizzare nella forma che abbiamo appena visto, poiché questa viola il principio di minima sorpresa, dato che l'attributo è nascosto anche all'interno della classe:

```
>>> class Foo(metaclass=ChangeHiddenMeta):
...     __a = 33
...     def foo(self):
...         print(Foo.__a)
...
>>> f = Foo()
>>> f.foo()
Traceback (most recent call last):
...
AttributeError: type object 'Foo' has no attribute '_Foo__a'
```

Il comportamento che si dovrebbe avere, come abbiamo spiegato nel [Capitolo 5](#), è questo:

```
>>> class Foo:
...     __a = 33
...     def foo(self):
...         print(Foo.__a)
...
>>> f = Foo()
>>> f.foo()
33
>>> Foo.__a
Traceback (most recent call last):
...
AttributeError: type object 'Foo' has no attribute '__a'
```

Come utile esercizio possiamo riscrivere autonomamente l'esempio di questa sezione, in modo da non violare il principio di minima sorpresa.

Implementazione del singleton pattern con le metaclassi

Nella sezione *La creazione delle istanze* di questo capitolo abbiamo visto una implementazione del singleton pattern, che consiste nel fare l'overriding del metodo `__new__()` in modo che questo crei una istanza della classe quando non ne è stata già creata una in precedenza, altrimenti restituisca l'istanza precedentemente creata:

```
>>> class Foo:
...     def __new__(cls):
```

```

... if not hasattr(cls, '_instance'):
...     cls._instance = super().__new__(cls)
...     return cls._instance
...
>>> f1 = Foo()
>>> f2 = Foo()
>>> f1 is f2
True

```

Possiamo ottenere lo stesso risultato con le metaclassi:

```

>>> class SingletonMeta(type):
...     def __call__(cls, *args, **kwargs):
...         if not hasattr(cls, '_instance'):
...             cls._instance = super().__call__(*args, **kwargs)
...         return cls._instance
...
>>> class Foo(metaclass=SingletonMeta):
...     pass
...
>>> f1 = Foo()
>>> f2 = Foo()
>>> f1 is f2
True

```

Questo è ciò che accade: la chiamata alla classe `Foo` viene tradotta in `type(Foo).__call__(Foo)`, ovvero in `SingletonMeta.__call__(Foo)`, la quale crea una nuova istanza di `Foo` solo se non ne è stata creata una in precedenza.

Approfondiamo con qualche osservazione: abbiamo fatto in modo che `SingletonMeta.__call__()` dopo il primo argomento (la classe) accetti un numero variabile di argomenti, compresi quelli con matching per nome. Questo perché non sappiamo come verranno definite le classi di tipo `SingletonMeta`, per cui non conosciamo a priori il numero e la modalità di matching dei loro argomenti:

```

>>> class Foo(metaclass=SingletonMeta):
...     def __init__(self, a, b, c=33):
...         self.a = a
...         self.b = b
...         self.c = c
...
>>> f = Foo(b=22, a=11)
>>> f.__dict__
{'a': 11, 'c': 33, 'b': 22}

```

In questo caso, ad esempio, `Foo(b=22, a=11)` corrisponde a `SingletonMeta.__call__(Foo, b=22, a=11)`.

Infine, analizziamo l'istruzione `cls._instance = super().__call__(*args, **kwargs)`. Come abbiamo detto nel [Capitolo 5](#), a partire da Python 3 è possibile chiamare la classe `super` senza argomenti. Nel nostro caso la chiamata precedente viene tradotta in `super(SingletonMeta, Foo).__call__(*args, **kwargs)`, e `super` delega la chiamata al metodo `__call__()` della classe che, nel MRO di `SingletonMeta`, segue `SingletonMeta`. Questa classe è `type`:

```
>>> SingletonMeta.__mro__
(<class '__main__.SingletonMeta'>, <class 'type'>, <class 'object'>)
```

Quindi, in definitiva, la chiamata `super(SingletonMeta, Foo).__call__(*args, **kwargs)` viene tradotta nella chiamata `type.__call__(Foo, *args, **kwargs)` e quest'ultima, come abbiamo detto nella sezione *La creazione delle istanze* di questo capitolo, si occupa di chiamare il metodo `Foo.__new__()` ed eventualmente `Foo.__init__()`. La chiamata `Foo.__new__()` corrisponde a `type.__new__()`, perché di quest'ultimo non è stato fatto l'overriding in nessuna classe che viene prima di `type` nel MRO di `Foo`, mentre del metodo `type.__init__()` è stato fatto l'overriding in `Foo`, per cui viene chiamato proprio `Foo.__init__()`.

Impedire l'istanziamento di classi omonime

Vediamo un altro esempio che ci consente di giocare ancora un po' con le metaclassi. Anche se non ha una grande valenza pratica, ci permette di fare ulteriore esercizio con poche righe di codice.

Consideriamo l'esempio seguente:

```
>>> class Foo:
...     a = 33
...
>>> id(Foo)
39434816
>>> class Foo:
...     b = 44
...
>>> id(Foo)
39442256
```

La prima istruzione `class` crea la classe con identificativo `39434816` e la assegna all'etichetta `Foo`. Come sappiamo, la seconda istruzione `class` crea una nuova classe e la assegna, ancora una volta, all'etichetta `Foo`, che prima faceva

riferimento alla prima classe creata. La memoria viene quindi liberata dalla prima classe, poiché non vi sono più etichette che fanno riferimento a essa. Consideriamo ora la seguente metaclassa:

```
>>> class NoHomonymMeta(type):
...     _names = {}
...     def __new__(metacls, name, *args, **kwargs):
...         if not name in metacls._names:
...             metacls._names[name] = super().__new__(metacls, name, *args, **kwargs)
...         else:
...             raise Exception("Una classe chiamata %s esiste già" % name)
...         return metacls._names[name]
```

Ecco cosa accade se proviamo a creare una classe di tipo `NoHomonymMeta`, dopo che ne è stata istanziata una con medesimo nome:

```
>>> class Foo(metaclass=NoHomonymMeta):
...     pass
...
>>> class Foo(metaclass=NoHomonymMeta):
...     pass
...
Traceback (most recent call last):
...
Exception: Una classe chiamata Foo esiste già
```

La metaclassa `NoHomonymMeta` fa l'overriding del costruttore della sua classe base (`type.__new__()`), in modo da verificare, per ogni classe che si vuole istanziare, se ne è stata già istanziata una con lo stesso nome, nel qual caso solleva una eccezione.

Ordine di definizione degli attributi nell'istruzione class

Consideriamo la classe seguente:

```
>>> class Foo:
...     tre = 3
...     quattro = 4
...     cinque = 5
```

Non abbiamo modo di conoscere l'ordine con cui gli attributi sono stati definiti nell'istruzione `class`, perché il namespace è implementato tramite un

dizionario e, come sappiamo, questo non è ordinato rispetto all'ordine di inserimento degli elementi:

```
>>> d = {}
>>> d['tre'] = 3
>>> d['quattro'] = 4
>>> d['cinque'] = 5
>>> [name for name in d]
['quattro', 'tre', 'cinque']
```

Infatti:

```
>>> [name for name in Foo.__dict__ if not name.startswith('_')]
['quattro', 'tre', 'cinque']
```

Se vogliamo tenere traccia dell'ordine con cui sono definiti gli attributi all'interno dell'istruzione `class`, possiamo banalmente optare per la seguente soluzione:

```
>>> class Foo:
...     tre = 3
...     quattro = 4
...     cinque = 5
...     _order = ('tre', 'quattro', 'cinque')
...
>>> Foo._order
('tre', 'quattro', 'cinque')
```

Questa non è di certo una grandissima idea, perché dobbiamo manualmente indicare l'ordine, e gli svantaggi sono notevoli. Ad esempio, ogni volta che decidiamo di cambiare l'ordine di definizione degli attributi, dobbiamo modificare anche l'attributo `_order`, indicando il nuovo ordine. Se dobbiamo fare questa operazione per una sola classe, non è un grosso problema, ma lo diventa se le classi da modificare sono decine o centinaia.

Una soluzione migliore, però, esiste: creare una metaclassa che definisca per noi l'attributo `_order` al momento della creazione della classe. Per fare ciò non dobbiamo fare altro che far restituire a `__prepare__()` un dizionario ordinato e creare l'attributo `_order` in `__new__()`, assegnandogli le chiavi ordinate del dizionario preso da `__new__()` come argomento.

Se facciamo memoria locale, ricorderemo che nel [Capitolo 2](#) abbiamo parlato del tipo `collections.OrderedDict`, introdotto con Python 3.1, il quale istanzia un dizionario ordinato:

```
>>> import collections
>>> d = collections.OrderedDict()
>>> d['tre'] = 3
>>> d['quattro'] = 4
>>> d['cinque'] = 5
>>> tuple(d)
('tre', 'quattro', 'cinque')
```

La soluzione è, quindi, immediata:

```
>>> import collections
>>> class OrderMeta(type):
...     def __prepare__(clsname, bases):
...         return collections.OrderedDict()
...     def __new__(metacls, name, bases, namespace):
...         cls = super().__new__(metacls, name, bases, namespace)
...         cls._order = (name for name in namespace if not name.startswith('__'))
...         return cls
...
>>> class Foo(metaclass=OrderMeta):
...     tre = 3
...     quattro = 4
...     cinque = 5
...
>>> tuple(Foo._order)
('tre', 'quattro', 'cinque')
```

Come possiamo vedere, abbiamo deciso di ordinare solamente gli attributi che non iniziano con un doppio underscore.

Metaclassi che accettano argomenti opzionali

Quando si definisce una classe, è possibile passare alla metaclassa degli argomenti opzionali. Questi argomenti devono essere passati nella clausola `class` con modalità di matching keyword-only. Ovviamente i metodi `__prepare__()`, `__new__()` e `__init__()` della metaclassa devono accettare questi argomenti:

```
>>> class FooMeta(type):
...     def __prepare__(clsname, bases, *, a, b):
...         print('In FooMeta.__prepare__(): a -> %s; b -> %s' %(a, b))
...         return {}
...     def __new__(metacls, name, bases, namespace, *, a, b):
...         print('In FooMeta.__new__(): a -> %s; b -> %s' %(a, b))
...         return super().__new__(metacls, name, bases, namespace)
```

```

... def __init__(self, name, bases, namespace, *, a, b):
...     print('In FooMeta.__init__(): a -> %s; b -> %s' %(a, b))
...     return super().__init__(name, bases, namespace)
...
>>> class Foo(metaclass=FooMeta, a=33, b=44):
...     pass
...
In FooMeta.__prepare__(): a -> 33; b -> 44
In FooMeta.__new__(): a -> 33; b -> 44
In FooMeta.__init__(): a -> 33; b -> 44

```

Test driven development

Il *test driven development* (TDD) è un processo di sviluppo del software in cui la scrittura del codice che implementa le funzionalità è preceduta e guidata (driven) dalla scrittura dei relativi test di validazione. Sostanzialmente il TDD si articola nelle seguenti fasi:

1. si scrive sia l'interfaccia del codice da validare sia il relativo test;
2. si esegue il test con lo scopo che fallisca; infatti, se non vi sono errori nel test, allora questo deve fallire, visto che vi sono solo le interfacce del codice da validare e manca l'implementazione;
3. si implementa una versione minimale del codice da validare, che consenta di superare i test;
4. si esegue il test: se non viene superato, allora si cerca di individuare la causa del fallimento, per poi eseguire nuovamente il test; se il test viene superato, si passa al punto 5;
5. si ristruttura il codice (si fa il *refactoring*), in modo da migliorarlo il più possibile, ad esempio rendendolo più robusto, più leggibile, più veloce, rivedendo la documentazione ecc. Dopo ogni miglioria introdotta, vanno rieseguiti i test in modo da verificare immediatamente che con il refactoring non siano stati introdotti dei bug.

Il flusso di lavoro del test driven development

I cinque punti che abbiamo appena visto normalmente vengono riassunti in modo ancora più conciso:

1. scrivere il test e accertarsi che fallisca;
2. implementare una versione minimale del codice, che consenta di superare il test;
3. effettuare il refactoring.

Vediamo un semplice esempio per capire meglio ciò di cui stiamo parlando. Supponiamo di dover scrivere una funzione, che chiamiamo `adder()`, la quale prende due oggetti di tipo qualunque che possono essere sommati tra loro e ne restituisce la somma.

Scrivere il test e accertarsi che fallisca

Definiamo l'interfaccia della nostra funzione:

```
$ cat adder.py
def adder(a, b):
    pass
```

Scriviamo un test che consenta di verificare la correttezza delle funzionalità richieste alla funzione:

```
$ cat adder_test.py
from adder import adder
if __name__ == '__main__':
    assert ['a', 'b'] != adder(['a'], ['b']), 'Somma di due liste: test non superato!'
    assert 3 != adder(1, 2), 'Somma di due interi: test non superato!'
    assert 'python3' != adder('python', '3'), 'Somma di due stringhe: test non superato!'
```

A questo punto siamo pronti a eseguire il test, in modo da accertarci che fallisca. Questa verifica, anche se può sembrare inutile, in realtà è veramente importante. Se il test non fallisce, significa che abbiamo commesso qualche errore nello scriverlo: dovrebbe fallire, infatti, dal momento che le funzionalità della funzione `adder()` non sono state ancora implementate.

Procediamo, quindi, con la verifica:

```
$ python adder_test.py
$
```

Il test è stato superato (non viene mostrato alcun messaggio di errore) e questo significa che abbiamo commesso degli errori nello scriverlo. Infatti, se prestiamo più attenzione, noteremo che nell'istruzione `assert` abbiamo usato il simbolo `!=` piuttosto che `=`.

Correggiamo, quindi, gli errori:

```
$ cat adder_test.py
from adder import adder
if __name__ == '__main__':
    assert ['a', 'b'] == adder(['a'], ['b']), 'Somma di due liste: test non superato!'
    assert 3 == adder(1, 2), 'Somma di due interi: test non superato!'
    assert 'python3' == adder('python', '3'), 'Somma di due stringhe: test non superato!'
```

e, ancora una volta, eseguiamo il test con lo scopo di accertarci che fallisca:

```
$ python adder_test.py
Traceback (most recent call last):
...
AssertionError: Somma di due liste: test non superato!
```

Il test è fallito e questo ci rassicura sul fatto che fa bene il suo mestiere. Adesso siamo pronti per implementare la funzione `adder()`.

Implementare una versione minimale del codice che consenta di superare il test

Implementiamo la funzione `adder()`:

```
$ cat adder.py
def adder(a, b):
    return a + b
```

Verifichiamo che il test passi:

```
$ python adder_test.py
$
```

Benissimo, il test è stato superato al primo tentativo, per cui possiamo passare alla fase successiva: il *refactoring*.

Effettuare il refactoring del codice

Il codice della funzione `adder()` è banale e sostanzialmente non c'è da lavorare sul refactoring: non abbiamo nulla da ottimizzare e non vi sono migliorie da apportare per renderlo più robusto o leggibile. Il nostro refactoring si riduce, quindi, alla scrittura della documentazione:

```
$ cat adder.py
def adder(a, b):
    """Restituisci la somma di due oggetti."""
    return a + b
```

Completato il refactoring, eseguiamo nuovamente il test, in modo da verificare che non siano stati introdotti degli errori:

```
$ python adder_test.py
Traceback (most recent call last):
...
SyntaxError: EOF while scanning triple-quoted string literal
```

Il test non è stato superato. Scrivendo la docstring, infatti, ci siamo dimenticati di un apice doppio, per cui essa termina con soli due apici doppi piuttosto che con tre. Quindi con il refactoring abbiamo introdotto un errore. Correggiamo, aggiungendo il terzo apice doppio:

```
$ cat adder.py
def adder(a, b):
    """Restituisci la somma di due oggetti."""
    return a + b
```

Eseguiamo nuovamente il test:

```
$ python adder_test.py
$
```

Il test è stato superato, per cui il nostro lavoro è concluso.

I test automatici

Il TDD non si basa sull'esecuzione di test manuali, ma sulla scrittura e la successiva esecuzione di *test automatici*. Vediamo di chiarire meglio questo concetto. Se non avessimo scritto il file *adder_test.py*, allora a ogni esecuzione del test avremmo dovuto importare a mano la funzione `adder()` e scrivere una per una le istruzioni `assert`:

```
>>> from adder import adder
>>> assert ['a', 'b'] == adder(['a'], ['b'])
>>> assert 3 == adder(1, 2)
>>> assert 'python3' == adder('python', '3')
```

Dopo qualche ciclo di test ci sarebbe passata la voglia di praticare il test driven development... Il nostro obiettivo è quello di poter eseguire i test in modo veloce e, per poter fare ciò, dobbiamo scrivere del codice che in modo automatico esegua le verifiche. Questo codice viene chiamato, per l'appunto, *test automatico*.

Una caratteristica dei test automatici è di essere *ripetibili*. Questa condizione non è garantita quando eseguiamo i test in modo manuale. Supponiamo, infatti, di scrivere dei test a mano, con lo scopo di verificare 10 funzionalità (per non dire 100...): oltre a perdere tantissimo tempo, a ogni esecuzione dei test corriamo il rischio sia di dimenticare qualche condizione da testare, sia di scriverla in modo diverso (e magari anche sbagliato) rispetto a quanto avevamo fatto nel precedente ciclo di test; questi inconvenienti accadono abbastanza di frequente quando si eseguono i test manualmente e intercorre molto tempo (settimane, mesi, anni) tra una sessione di verifica e l'altra.

Il fatto che possa intercorrere molto tempo tra due sessioni di verifica consecutive non deve stupire. Questo può accadere, infatti, sia perché dopo che il software è andato in produzione vengono individuati nuovi bug, sia

perché il software evolve e spesso è necessario renderlo più robusto, implementare nuove funzionalità o modificare quelle esistenti, cambiare le interfacce, apportare ottimizzazioni ecc.

L'importanza dei test automatici

Pensiamo al seguente scenario: abbiamo speso vari mesi di lavoro per scrivere del software e abbiamo eseguito una serie di test, manualmente, che ci hanno rassicurati sul fatto che tutto funziona a dovere. Il software va in produzione e, come succede (sempre), dopo N mesi viene individuato un bug. In questi N mesi abbiamo fatto tanti altri lavori e non ci ricordiamo in dettaglio come funziona il nostro software, per cui dedichiamo del tempo a rinfrescarci le idee e poi iniziamo a lavorare al nostro bug, intervenendo ovviamente con delle modifiche al codice. Fatto ciò, iniziamo a verificare che il problema sia risolto.

Bene, dopo un po' di lavoro, sembra che abbiamo risolto e sistemato il bug... Ma cosa possiamo dire su tutte le altre funzionalità? Questo intervento sul codice ha risolto un problema, ma, ovviamente, potrebbe averne causati altri. Per essere certi che tutto funzioni, dovremmo ripensare a tutti i possibili scenari e test di validazione che abbiamo eseguito N mesi prima e ripeterli tutti, uno per uno. Magari dobbiamo pensare a tutto ciò con il fiato sul collo, del nostro capo o dei nostri clienti, perché, come abbiamo detto, sono passati N mesi da quando non lavoriamo più a questo software e nel frattempo abbiamo iniziato a lavorare ad altri progetti e abbiamo nuove scadenze. È uno scenario che fa venire i brividi!

Quando lavoravamo full-time a questo progetto e avevamo chiari in mente tutti i dettagli del software, avremmo impiegato poco tempo per scrivere i test automatici e sarebbe stato un grandissimo investimento, perché adesso e in futuro avremmo potuto fare tutte le verifiche del caso, in pochi secondi, con un semplice:

```
$ python mymodule_test.py
```

Inoltre, pensiamo che la realtà è molto peggiore, poiché lo scenario appena descritto non è un caso isolato, ma si ripete M volte nel tempo, dove M è la somma tra il numero di bug che verranno individuati e le nuove modifiche che per varie ragioni dovranno essere apportate al software.

Detto ciò, non credo ci sia altro da aggiungere per convincerci di quanto sia importante scrivere i test automatici.

Le tipologie di test

I test automatici vengono tipicamente classificati nel modo seguente:

1. *test unitari (unit testing)*: servono per verificare il corretto funzionamento delle *unità software*, ovvero di quelle porzioni di codice le cui funzionalità possono essere verificate senza la necessità di dover interagire con altre parti del sistema;
2. *test di integrazione (integration testing)*: servono per verificare che le varie unità software siano integrate correttamente nel sistema;
3. *test di sistema (system testing)*: servono per verificare che il sistema nel suo complesso funzioni come ci si aspetta;
4. *test di accettazione (acceptance testing)*: servono per verificare che il sistema si comporti come il cliente si aspetta. È quindi il cliente stesso che esegue i test e decide se questi possono essere considerati superati o meno.

Ovviamente, come ci si può immaginare, ci sono tante sfumature che fanno sì che un test possa essere considerato appartenente a più di una categoria. Per quanto ci riguarda, non siamo interessati ad approfondire i dettagli relativi a questa suddivisione, poiché lo scopo dell'esercizio è quello di imparare a scrivere i test unitari automatici e, più in generale, la pratica del test driven development. Se si vogliono approfondire le differenze tra le varie tipologie di testing, un ottimo punto di partenza è Wikipedia: http://en.wikipedia.org/wiki/Software_testing.

I test unitari automatici

Cerchiamo di chiarire meglio il concetto di test unitario. A tale scopo consideriamo ancora la funzione `adder()` vista in precedenza. Questa è una unità software, poiché è una porzione di codice che può essere validata senza dover interagire con altre porzioni di codice. Il test che abbiamo scritto per verificare il corretto funzionamento di `adder()` è quindi un test unitario.

Tipicamente un test unitario riguarda la verifica di vari requisiti dell'unità software. Ad esempio, se alla funzione `adder()` è richiesto di eseguire la somma di due oggetti di tipo qualsiasi e anche di sollevare una eccezione di tipo `TypeError` nel caso non sia possibile eseguire la somma, allora dobbiamo verificare sia che `adder()` esegua la somma in modo corretto, come abbiamo fatto, sia che sollevi una eccezione quando la somma dei due oggetti non è possibile. Quest'ultima condizione non può essere verificata con l'istruzione `assert`, ma dobbiamo inventarci qualcos'altro, come ad esempio una gestione dell'eccezione:

```
$ cat test_adder.py
from adder import adder
if __name__ == '__main__':
    assert ['a', 'b'] == adder(['a'], ['b']), 'Somma di due liste: test non superato!'
    assert 3 == adder(1, 2), 'Somma di due interi: test non superato!'
    assert 'python3' == adder('python', '3'), 'Somma di due stringhe: test non superato!'
    try:
        adder(1, '2')
    except TypeError:
        pass
    else:
        print('Somma di un intero e una stringa: test non superato!')
```

Questo test viene superato:

```
$ python test_adder.py
$
```

È evidente che, all'aumentare del numero delle funzionalità richieste e della complessità del codice da validare, la scrittura dei test unitari diventa sempre più impegnativa e si corre il rischio di:

- scrivere del codice di verifica non appropriato: in questo caso il test potrebbe non rilevare gli errori e questo non significa che il test è inutile, ma molto peggio: significa che il test è pericoloso, perché se non rivela gli errori restiamo convinti che il programma funzioni correttamente;
- scrivere dei test troppo complessi: se il codice di validazione non è semplice, allora aumentano le probabilità che vi siano errori nel test e quindi dovremmo scrivere un test per validare il corretto funzionamento del test...;
- perdere più tempo nel far funzionare correttamente il test piuttosto che nel far funzionare correttamente il codice oggetto della verifica.

Per questi e altri motivi, i test unitari vanno scritti utilizzando dei framework creati appositamente per facilitare la scrittura dei test. Ci si potrebbe chiedere come un framework possa essere così generico da consentire di rilevare tutte le possibili condizioni di errore. La risposta è semplice: qualunque sia l'unità software che si vuole validare, nella maggior parte dei casi le condizioni da verificare sono delle *asserzioni* che ricadono all'interno di un insieme molto piccolo: test di uguaglianza (*assert equal*), test di quasi uguaglianza (*assert almost equal*), test di verifica di sollevamento di un certo tipo di eccezione (*assert raises*) ecc. Questo insieme di asserzioni, nonostante sia piccolo, tipicamente consente di verificare la totalità, o quasi, delle specifiche che l'unità software sotto test deve rispettare.

NOTA

Quando un test viene superato, non significa che il software sotto validazione funzioni perfettamente e sia esente da bug. Lo unit testing, infatti, come ogni altra forma di testing, non può assicurare l'assenza di errori, ma è utile per evidenziarne la presenza.

Come sappiamo, Python ha le batterie incluse, per cui fornisce il supporto anche per la scrittura di test unitari. Il modulo da utilizzare è `unittest`.

Il modulo `unittest` della libreria standard

Il modulo `unittest` della libreria standard è un framework che fornisce gli strumenti per scrivere in modo semplice i test unitari. Vediamo di utilizzarlo per riscrivere `test_adder.py`:

```
$ cat test_adder.py
import unittest
from adder import adder

class AdderTest(unittest.TestCase):
    def test_addition(self):
        """Verifica che adder() esegua correttamente la somma."""
        self.assertEqual(['a', 'b'], adder(['a'], ['b']))
        self.assertEqual(3, adder(1, 2))
        self.assertEqual('python3', adder('python', '3'))
    def test_exception(self):
        """Verifica che adder() sollevi una eccezione quando la somma non è supportata."""
        self.assertRaises(TypeError, adder, 1, '2')

if __name__ == '__main__':
    unittest.main()
```

Come possiamo vedere, è tutto molto semplice:

- abbiamo scritto una classe `AdderTest` che eredita da `unittest.TestCase`;
- ciascun metodo che inizia con `test` contiene il codice di verifica di una specifica funzionalità;
- le verifiche vengono fatte utilizzando i metodi `assert*` definiti dal framework (ereditati dalla classe base `unittest.TestCase`), per cui non dobbiamo ingegnarci per scrivere del codice di nostro pugno;
- abbiamo chiamato la funzione `unittest.main()`, detta *test runner*, la quale si occupa di eseguire i metodi il cui nome inizia con `test`, uno per volta e in

modo *non ordinato*.

Tra breve analizzeremo nel dettaglio il codice. Per il momento soffermiamoci a vedere cosa accade quando eseguiamo il test:

```
$ python test_adder.py
```

```
..
```

```
-----  
Ran 2 tests in 0.000s
```

```
OK
```

Veniamo informati del fatto che sono stati eseguiti due test e che tutto è andato a buon fine, nel senso che sono stati superati. Se vogliamo avere un output più completo, possiamo utilizzare lo switch `-v`:

```
$ python test_adder.py -v
```

```
test_addition (__main__.AdderTest)
```

```
Verifica che adder() esegua correttamente la somma. ... ok
```

```
test_exception (__main__.AdderTest)
```

```
Verifica che adder() sollevi una eccezione quando la somma non è supportata. ... ok
```

```
-----  
Ran 2 tests in 0.000s
```

```
OK
```

Vediamo cosa accade quando il test non passa. A tale scopo modifichiamo il file di test sostituendo la chiamata `self.assertEqual(3, adder(1, 2))` con `self.assertEqual(2, adder(1, 2))`:

```
$ python test_adder.py
```

```
F.
```

```
=====
```

```
FAIL: test_addition (__main__.AdderTest)
```

```
Verifica che adder() esegua correttamente la somma.
```

```
-----  
Traceback (most recent call last):
```

```
File "adder_test.py", line 8, in test_addition
```

```
self.assertEqual(2, adder(1, 2))
```

```
AssertionError: 2 != 3
```

```
-----  
Ran 2 tests in 0.000s
```

```
FAILED (failures=1)
```

Il test giustamente fallisce e viene indicato anche il motivo. Ora siamo pronti per analizzare il codice nel dettaglio e approfondire l'utilizzo del modulo `unittest`.

I casi di test

Per validare le unità software si utilizzano i cosiddetti *test case* (casi di test). Se vogliamo scrivere un test case, dobbiamo definire una classe che eredita da `unittest.TestCase`, come abbiamo fatto nel caso di `AdderTest`:

```
class AdderTest(unittest.TestCase):
    def test_addition(self):
    ...
```

In questo modo `AdderTest` eredita da `unittest.TestCase` i metodi `assert*` che utilizzeremo per effettuare le validazioni.

Il test runner e le asserzioni

Quando eseguiamo il test, la funzione `unittest.main()`, chiamata *test runner*, chiama uno per volta e in ordine casuale tutti i metodi del test case che iniziano con `test`, quindi, nel nostro caso, chiama `AdderTest.test_addition()` e `AdderTest.test_exception()`. Questi metodi eseguono la validazione tramite la verifica di alcune asserzioni. Come è facile intuire, le asserzioni sono verificate dai metodi che iniziano con `assert`, i quali vengono ereditati dalla classe base `unittest.TestCase`.

Nel nostro esempio abbiamo utilizzato `TestCase.assertEqual()` e `TestCase.assertRaises()`. Il metodo `assertEqual()` verifica che il suo primo argomento sia uguale al secondo argomento e se questa condizione non è verificata fa fallire il test:

```
>>> import unittest
>>> t = unittest.TestCase()
>>> t.assertEqual(1, 1)
>>> t.assertEqual(1, 2)
Traceback (most recent call last):
...
AssertionError: 1 != 2
```

Il metodo `assertRaises()` è invece più sofisticato: verifica che un oggetto callable, quando gli viene passato un dato set di argomenti, sollevi un certo tipo di eccezione. Quindi, nel nostro caso, la chiamata `assertRaises(TypeError, adder, 1, '2')` verifica che `adder(1, '2')` sollevi una eccezione di tipo `TypeError` e se questo non accade il test fallisce:

```
>>> t.assertRaises(TypeError, adder, 1, '2')
>>> t.assertRaises(TypeError, adder, 1, 2)
Traceback (most recent call last):
...
AssertionError: TypeError not raised by adder
```

Il metodo `TestCase.assertRaisesRegex()` è ancora più stringente di `assertRaises()`, perché

verifica che nel messaggio di errore vi sia del testo che combacia con l'espressione regolare passata come argomento. Ad esempio, nel caso seguente viene verificato che l'espressione regolare `u*supported` combaci con del testo presente nel messaggio di errore:

```
>>> t.assertRaisesRegex(TypeError, 'u*supported', adder, 1, '2')
>>>
```

Il test viene superato poiché nel messaggio di errore è presente la parola `unsupported`, che combacia con l'espressione regolare. Nel caso seguente, invece, l'espressione regolare `p*thon` non combacia con alcuna porzione di testo nel messaggio di errore, per cui il test non viene superato:

```
>>> t.assertRaisesRegex(TypeError, 'p*thon', adder, 1, '2')
Traceback (most recent call last):
...
AssertionError: "p*thon" does not match "unsupported operand type(s) for +: 'int' and 'str'"
```

Oltre a queste appena viste, la classe `TestCase` definisce anche altre asserzioni, che troviamo elencate nella documentazione online, alla pagina web <http://docs.python.org/3/library/unittest.html>. Molte di esse sono state introdotte a partire da Python 3.1.

Le test fixture

Quando si esegue un test case, spesso è utile poter compiere una serie di azioni propedeutiche al test, come ad esempio l'apertura di file, di socket, o semplicemente la creazione di una istanza sulla quale effettuare i test, o altro ancora. Allo stesso modo, spesso è utile poter compiere delle azioni al termine di ogni test, come la chiusura di file, di socket ecc. Queste azioni di setup e finalizzazione sono dette *test fixture*. Se definiamo i metodi `setUp()` e `tearDown()`, allora questi vengono chiamati automaticamente da `unittest.main()` per compiere le test fixture. Il metodo `setUp()` viene chiamato prima dell'esecuzione di ciascun test, mentre il metodo `tearDown()` al termine di ogni test. Consideriamo, ad esempio, il seguente file di test:

```
$ cat foo.py
import unittest
class FooTest(unittest.TestCase):
    def setUp(self):
        print('In setUp()')
    def tearDown(self):
        print('In tearDown()')
```

```
def test_1(self):
print('Esecuzione di test_1() ...')
def test_2(self):
print('Esecuzione di test_2() ...')
if __name__ == '__main__':
unittest.main()
```

Ecco cosa accade quando lo eseguiamo:

```
$ python foo.py
In setUp()
Esecuzione di test_1() ...
In tearDown()
In setUp()
Esecuzione di test_2() ...
In tearDown()
.
```

```
-----
Ran 2 tests in 0.000s
OK
```

Test discovery

A partire da Python 3.2, è possibile delegare al modulo `unittest` la ricerca e la successiva esecuzione dei test. Per fare ciò dobbiamo semplicemente eseguire il modulo `unittest` da linea di comando. Ad esempio, poniamoci all'interno della directory contenente il file *adder.py* e il file *test_adder.py*:

```
$ ls
adder.py test_adder.py
```

Se adesso diamo il comando `python -m unittest`, allora `unittest` cerca all'interno della directory corrente i moduli importabili che iniziano per `test` e poi li importa, per cui vengono eseguiti:

```
$ python -m unittest
```

```
..
```

```
-----
Ran 2 tests in 0.000s
OK
```

In questo caso `unittest` ha trovato ed eseguito il modulo `test_adder` e quindi sono stati eseguiti i due metodi `test_addition()` e `test_exception()`. Si osservi che `unittest` esegue tutti i file con estensione `.py`, a prescindere dal fatto che contengano test unitari o no:

```
$ echo "print('FILE FOO')" > test_foo.py
```

```
$ ls
adder.py test_adder.py test_foo.py
$ python -m unittest
FILE FOO
..
-----
Ran 2 tests in 0.000s
OK
```

Il test discovery può essere personalizzato in vari modi. Ancora una volta, per maggiori informazioni rimandiamo alla documentazione ufficiale del modulo `unittest`.

NOTA

Se non siamo soddisfatti da `unittest`, possiamo provare un altro framework di terze parti chiamato *nose*, che è una estensione del modulo `unittest` che facilita ulteriormente la scrittura dei test automatici. Per maggiori informazioni su *nose*: <http://nose.readthedocs.org/en/latest/>.

Confronto tra unit testing e docstring validation testing

Il docstring validation testing, di cui abbiamo parlato nel [Capitolo 4](#), e lo unit testing, di cui stiamo parlando in questo capitolo, hanno scopi ben differenti. Il primo serve per validare gli esempi presenti nelle docstring, il secondo per validare il codice.

Quindi lo scopo del modulo `doctest` è quello di fornire gli strumenti per validare gli esempi presenti nella documentazione e non di validare il codice vero. Se gli esempi sono riportati correttamente e si apportano modifiche al codice tali per cui l'esecuzione del test con `doctest` fallisce, allora il DVT ci consente di rilevare l'errore, ma questo è solo un effetto secondario e non significa che lo scopo del DVT è quello di rilevare gli errori nel codice.

Sono solo i test unitari scritti durante il processo del test driven development a essere realizzati con lo scopo di validare le unità software, e non potrebbe essere diversamente perché, se ci pensiamo, gli esempi riportati nella documentazione non vengono scritti prima che il codice funzioni, con lo scopo di validarlo, ma dopo che il codice è stato validato, copiando l'esempio dalla shell interattiva per poi incollarlo e documentarlo nella docstring.

Per questo motivo, lo unit testing e il docstring validation testing non sono due modalità di testing mutuamente esclusive, anzi, sarebbe bene combinarle assieme all'interno del processo di test driven development, ad esempio nel

modo seguente:

1. scrivere i test unitari, accertandosi che falliscano;
2. implementare una versione minimale del codice che consenta di superare i test;
3. effettuare il refactoring occupandosi, tra le varie cose, di aggiungere nelle docstring eventuali esempi fatti utilizzando la shell interattiva (opportunamente commentati), aggiungendo nel modulo la chiamata `doctest.testmod()`;
4. eseguire il modulo in modo che vengano fatte le verifiche con `doctest`;
5. eseguire i test unitari, in modo che venga validato il codice.

Nella prossima sezione vedremo un esempio pratico di utilizzo di questo flusso di lavoro.

Esempio pratico di utilizzo del test driven development

In questa sezione analizzeremo un esempio di utilizzo congiunto di metaclassi e descriptor e metteremo in pratica quanto appreso nella sezione precedente sul test driven development. Inizieremo con il vedere il contenuto dei file, per poi spiegare passo passo la teoria e analizzare il significato del codice. Il primo step consisterà nello specificare i requisiti che deve avere il software, per poi adottare il test driven development al fine di implementare il codice e i relativi test unitari. Il risultato finale consiste nei seguenti file:

1. *typedesc.py*: definisce la classe `TypeDesc`, che è un descriptor;
2. *test_typedesc.py*: implementa i test unitari da eseguire per validare il descriptor;
3. *orderedtypemeta.py*: definisce una metaclassa chiamata `OrderedTypeMeta`;
4. *test_orderedtypemeta.py*: implementa i test unitari da eseguire per validare la metaclassa.

Implementazione e validazione del descriptor

Supponiamo di voler creare una classe `Book`, le cui istanze devono avere tre attributi: `title`, che rappresenta il titolo del libro, `author`, che rappresenta l'autore, e `year`, che rappresenta l'anno di pubblicazione:

```
>>> class Book:
...     def __init__(self, title, author, year):
...         self.title, self.author, self.year = title, author, year
...
>>> b = Book('Programmare con Python', 'Marco Buttu', 2014)
>>> b.title
'Programmare con Python'
>>> b.author
'Marco Buttu'
>>> b.year
2014
```

Supponiamo che gli attributi delle istanze della classe `Book` vengano poi salvati come campi di un database, per cui il loro tipo non può essere qualsiasi, ma deve essere coerente con la struttura dei campi del database. Ad esempio, supponiamo che i campi `title` e `author` del database siano delle stringhe e il campo `year` un intero. Vorremmo poter effettuare un controllo preventivo direttamente

con Python, al momento della creazione delle istanze, in modo che se, ad esempio, la classe viene istanziata nel modo seguente:

```
>>> b = Book('Programmare con Python', 'Marco Buttu', '2014')
>>> b.year
'2014'
```

o in quest'altro modo:

```
>>> import datetime
>>> b = Book('Programmare con Python', 'Marco Buttu', datetime.date(2014, 1, 1))
>>> b.year
datetime.date(2014, 1, 1)
```

venga sollevata una eccezione di tipo `TypeError` che informi che l'attributo `year` deve essere di tipo `int`. Questa è quindi la specifica che deve avere il nostro software.

Scrivere un test unitario e verificare che fallisca

Iniziamo il lavoro scrivendo una possibile interfaccia del descriptor:

```
$ cat typedesc.py
class TypeDesc:
def __init__(self, expected_type, value=None):
pass
def __set__(self, instance, value):
pass
```

L'obiettivo di questo punto è scrivere un test unitario e accertarsi che fallisca. Infatti, come abbiamo detto in precedenza, il codice da validare non è stato implementato e quindi non funziona, per cui, se il test non fallisce, significa che ci sono dei bug nel test stesso.

Prima di scrivere il test, chiariamoci le idee su ciò che vorremmo faccia il nostro descriptor. Consideriamo, a tale scopo, una classe `Book` analoga alla seguente:

```
>>> class Book:
... title = TypeDesc(str)
... author = TypeDesc(str)
... year = TypeDesc(int)
... def __init__(self, title, author, year):
... self.title, self.author, self.year = title, author, year
```

Ciò che vorremmo è che, una volta che `TypeDesc` è implementato correttamente, sollevi una eccezione quando si tenta di assegnare agli attributi di istanza `title`, `author` e `year` un oggetto di tipo diverso da quello previsto, analogamente a quanto è mostrato di seguito:

```
>>> b = Book('Programmare con Python', 'Marco Buttu', '2014')
Traceback (most recent call last):
...
TypeError: L'attributo `year` deve essere di tipo `int`
```

Scriviamo, quindi, un test unitario che consenta di validare queste specifiche:

```
$ cat test_typedesc.py
import unittest
from typedesc import TypeDesc
class TestBook(unittest.TestCase):
    def setUp(self):
        class Book:
            title = TypeDesc(str, 'title')
            author = TypeDesc(str, 'author')
            year = TypeDesc(int, 'year')
            def __init__(self, title, author, year):
                self.title, self.author, self.year = title, author, year
        self.Book = Book
        self.b = Book('Programmare con Python', 'Marco Buttu', 2014)
    def test_creation(self):
        """Verifica che durante la creazione venga sollevata una eccezione di tipo TypeError."""
        self.assertRaises(TypeError, self.Book, self.b.title, self.b.author, str(self.b.year))
    def test_setting(self):
        """Verifica che dopo la creazione gli assegnamenti avvengano in modo corretto."""
        title = 'I principi della matematica'
        author = 'Bertrand Russell'
        year = 1903
        self.b.title = title
        self.b.author = author
        self.b.year = year
        self.assertEqual(self.b.title, title)
        self.assertEqual(self.b.author, author)
        self.assertEqual(self.b.year, year)
    def test_wrongsetting(self):
        """Verifica che dopo la creazione un assegnamento errato sollevi una TypeError."""
        self.assertRaises(TypeError, setattr, self.b, 'title', [self.b.title])
        self.assertRaises(TypeError, setattr, self.b, 'author', [self.b.author])
        self.assertRaises(TypeError, setattr, self.b, 'year', str(self.b.year))
    def test_exceptmessage(self):
        """Verifica che il nome dell'attributo sia corretto."""
        self.assertRaisesRegex(TypeError, 'year', setattr, self.b, 'year', str(self.b.year))

if __name__ == '__main__':
    unittest.main()
```

Come possiamo vedere, il test case consiste di quattro metodi. Prima della

esecuzione di ciascuno di essi, viene eseguito il metodo `setUp()`, il quale crea una classe `Book` e una istanza di questa, ed entrambe vengono utilizzate nei test. Se abbiamo letto con attenzione la sezione relativa al test driven development, non dovremmo avere problemi nel capire il significato del test. In ogni caso, tutto sarà chiaro tra breve, appena inizieremo a implementare il codice del descriptor.

Detto ciò, prima di iniziare a lavorare all'implementazione, accertiamoci che il test fallisca:

```
$ python test_typedesc.py
FFFF
```

```
FAIL: test_creation (__main__.TestBook)
Verifica che durante la creazione venga sollevata una eccezione di tipo TypeError.
```

```
-----
Traceback (most recent call last):
...
AssertionError: TypeError not raised by Book
```

```
FAIL: test_exceptmessage (__main__.TestBook)
Verifica che il nome dell'attributo sia corretto.
```

```
-----
Traceback (most recent call last):
...
AssertionError: TypeError not raised by setattr
```

```
FAIL: test_setting (__main__.TestBook)
Verifica che dopo la creazione gli assegnamenti avvengano in modo corretto.
-----
...
AssertionError: <typedesc.TypeDesc object at 0x7fa324e69390> != 'I principi della matematica'
```

```
FAIL: test_wrongsetting (__main__.TestBook)
Verifica che dopo la creazione un assegnamento errato sollevi una TypeError.
```

```
-----
Traceback (most recent call last):
...
AssertionError: TypeError not raised by setattr
```

```
-----
Ran 4 tests in 0.001s
```

```
FAILED (failures=4)
```

Benissimo, sono falliti tutti e quattro i test, per cui possiamo andare avanti e iniziare a implementare il codice di `TypeDesc`.

Implementare le funzionalità richieste

Iniziamo con l'implementare le funzionalità che consentono di superare il primo test, chiamato `test_creation()`, che serve per verificare che durante la creazione dell'istanza venga sollevata una eccezione di tipo `TypeError` se uno dei tre argomenti non è istanza del tipo previsto:

```
$ cat typedesc.py
class TypeDesc:
    f __init__(self, expected_type, value=None):
    self._type, self.value = expected_type, value
    f __set__(self, instance, value):
    if isinstance(value, self._type):
    self.value = value
    else:
    raise TypeError("L'attributo deve essere di tipo '%s'" %self._type.__name__)
```

Eseguiamo nuovamente il test per capire se la funzionalità è stata implementata correttamente:

```
$ python test_typedesc.py
.FF.
```

```
FAIL: test_exceptmessage (__main__.TestBook)
Verifica che il nome dell'attributo sia corretto.
```

```
-----
Traceback (most recent call last):
```

```
...
AssertionError: "year" does not match "L'attributo deve essere di tipo `int`"
```

```
FAIL: test_setting (__main__.TestBook)
```

```
Verifica che dopo la creazione gli assegnamenti avvengano in modo corretto.
```

```
-----
Traceback (most recent call last):
```

```
...
AssertionError: <typedesc.TypeDesc object at 0x7f5b93c471d0> != 'I principi della matematica'
```

```
-----
Ran 4 tests in 0.001s
```

```
FAILED (failures=2)
```

Benissimo, sono stati superati sia `test_creation()` sia `test_wrongsettings()`. Quest'ultimo verifica che venga sollevata una eccezione di tipo `TypeError` se, dopo la creazione dell'istanza `b` di `Book`, si tenta di assegnare a `b.title`, `b.author` o `b.year` un oggetto di tipo diverso da quello previsto.

Dobbiamo ancora implementare le funzionalità relative agli altri due test. Iniziamo con `test_exceptmessage()`. Questo si aspetta che il nome dell'attributo compaia nel messaggio di errore, in modo che l'utente capisca dove sta il

problema. In sostanza vorremmo che il messaggio di errore sia, ad esempio, *L'attributo 'year' deve essere di tipo 'int' piuttosto che L'attributo deve essere di tipo 'int'*. La verifica viene fatta con il metodo `assertRaisesRegex()`, chiamandolo nel seguente modo:

```
self.assertRaisesRegex(TypeError, 'year', setattr, self.b, 'year', str(self.b.year))
```

In questo caso `assertRaisesRegex()` chiama `setattr(self.b, 'year')` e verifica, prima di tutto, che questo sollevi una eccezione `TypeError` e poi anche che nel messaggio di errore vi sia del testo che combaci con l'espressione regolare `'year'`. Per implementare questa funzionalità possiamo assegnare un nome all'attributo, nel seguente modo:

```
$ cat typedesc.py
class TypeDesc:
    f __init__(self, expected_type, name="", value=None):
        self._type, self._name, self.value = expected_type, name, value
    f __set__(self, instance, value):
        if isinstance(value, self._type):
            self.value = value
        else:
            raise TypeError("L'attributo '%s' deve essere di tipo '%s'"
                             %(self._name, self._type.__name__))
```

Eseguiamo il test:

```
$ python test_typedesc.py
..F.
```

```
FAIL: test_setting (__main__.TestBook)
Verifica che dopo la creazione gli assegnamenti avvengano in modo corretto.
-----
Traceback (most recent call last):
...
AssertionError: <typedesc.TypeDesc object at 0x7f60ed3f6110> != 'I principi della matematica'
-----
```

```
Ran 4 tests in 0.001s
```

```
FAILED (failures=1)
```

Anche questo test è stato superato. Manca ancora un test da superare: `test_setting()`.

Innanzitutto cerchiamo di capire perché il test non passa. Diamo uno sguardo al codice del test:

```
def test_setting(self):
    """Verifica che dopo la creazione gli assegnamenti avvengano in modo corretto."""
    title = 'I principi della matematica'
    author = 'Bertrand Russell'
    year = 1903
    self.b.title = title
    self.b.author = author
    self.b.year = year
    self.assertEqual(self.b.title, title)
    self.assertEqual(self.b.author, author)
    self.assertEqual(self.b.year, year)
```

Il messaggio di errore della `AssertionError` ci informa che:

```
AssertionError: <typedesc.TypeDesc object at 0x7f60ed3f6110> != 'I principi della matematica'
```

Questo significa che fallisce la prima asserzione. Vediamo di capire, concentrandoci sulla prima asserzione. Sostanzialmente abbiamo assegnato:

```
title = 'I principi della matematica'
self.b.title = title
```

Per cui ci aspettiamo che `self.b.title` sia uguale a `title` e che quindi la seguente asserzione non fallisca:

```
self.assertEqual(self.b.title, title)
```

Ecco cosa è successo: ci siamo dimenticati di implementare il metodo `__get__()` del descriptor, per cui `self.b.title` fa riferimento a `type(self.b).__dict__['title']`, ovvero all'istanza `TypeDesc(str, 'title')`:

```
>>> from typedesc import TypeDesc
>>> class Book:
...     title = TypeDesc(str, 'title')
...     author = TypeDesc(str, 'author')
...     year = TypeDesc(int, 'year')
...
>>> Book.__dict__['title']
<typedesc.TypeDesc object at 0x7f21d6cf0ed0>
```

Implementiamo, quindi, il metodo `TypeDesc.__get__()`:

```
$ cat typedesc.py
class TypeDesc:
    def __init__(self, expected_type, name="", value=None):
        self._type, self._name, self.value = expected_type, name, value
```

```
def __set__(self, instance, value):
    if isinstance(value, self._type):
        self.value = value
    else:
        raise TypeError("L'attributo '%s' deve essere di tipo '%s'"
            %(self._name, self._type.__name__))
def __get__(self, instance, instance_type):
    return self.value
```

Eseguiamo il test:

```
$ python test_typedesc.py
```

```
....
```

```
-----
```

```
Ran 4 tests in 0.000s
```

```
OK
```

Benissimo, il nostro software si comporta come ci aspettiamo. Adesso ci prendiamo del tempo per fare il refactoring.

Refactoring

Dando uno sguardo al codice, ci sembra che sia abbastanza leggibile e che, in questo momento, non vi siano migliorie da apportare, se non documentarlo meglio con degli esempi di codice funzionante, che possiamo copiare dalla shell interattiva. Ecco il risultato al termine di questo lavoro:

```
$ cat typedesc.py
```

```
"""Questo modulo definisce il descriptor `TypeDesc`.
```

La classe `TypeDesc` e' un descriptor che consente di indicare di che tipo deve essere un attributo di istanza. Ad esempio, consideriamo la seguente classe `Book`:

```
>>> class Book:
...     title = TypeDesc(str, 'title')
...     author = TypeDesc(str, 'author')
...     year = TypeDesc(int, 'year')
...     def __init__(self, title, author, year):
...         self.title, self.author, self.year = title, author, year
```

L'attributo di istanza `title` deve essere di tipo `str`, così come `author`, mentre `year` deve essere di tipo `int`. Quindi, se chiamiamo la classe passandole una stringa sia come primo sia come secondo argomento, e un intero come terzo argomento, l'istanza viene creata correttamente:

```
>>> b = Book('Programmare con Python', 'Marco Buttu', 2014)
>>> b.year
```

Se, invece, proviamo a istanziare la classe passandole dei tipi diversi da quelli attesi, allora viene sollevata una eccezione di tipo `TypeError`:

```
>>> b = Book('Programmare con Python', 'Marco Buttu', '2014')
Traceback (most recent call last):
```

```
....
TypeError: L'attributo `year` deve essere di tipo `int`
"""

class TypeDesc:
def __init__(self, expected_type, name="", value=None):
self._type, self._name, self.value = expected_type, name, value
def __set__(self, instance, value):
if isinstance(value, self._type):
self.value = value
else:
raise TypeError("L'attributo `%s` deve essere di tipo `%s`"
%(self._name, self._type.__name__))
def __get__(self, instance, instance_type):
return self.value
```

```
if __name__ == '__main__':
import doctest
doctest.testmod()
```

Abbiamo usato il modulo `doctest` in modo da validare le docstring. A questo punto effettuiamo subito il docstring validation testing, eseguendo il modulo:

```
$ python typedesc.py -v
...
4 passed and 0 failed.
Test passed.
```

Benissimo, gli esempi riportati nelle docstring funzionano correttamente. Rieseguiamo anche i test unitari, in modo da accertarci di non aver introdotto degli errori durante il refactoring:

```
$ python test_typedesc.py
....
-----
Ran 4 tests in 0.000s
OK
```

A questo punto il nostro lavoro in merito alla classe `TypeDesc` è concluso. Riassumendo:

- la classe `TypeDesc` funziona come ci aspettiamo, abbiamo validato il suo funzionamento con dei test automatici unitari, dei quali ci serviremo anche

- in futuro tutte le volte che apporteremo delle modifiche al codice;
- il file *typedesc.py* è stato documentato con utili esempi di funzionamento, validati con il modulo `doctest`. In futuro, tutte le volte che apporteremo dei cambiamenti al modulo, dovremo ricordarci di eseguire nuovamente il test della documentazione.

Passiamo adesso all'implementazione della metaclassa.

Implementazione e validazione della metaclassa

Vogliamo poter conoscere l'ordine con cui gli attributi di classe di tipo `TypeDesc` vengono definiti nella suite dell'istruzione `class`. Ad esempio, consideriamo una classe `Book` come la seguente:

```
>>> class Book:
...     title = TypeDesc(str, 'title')
...     author = TypeDesc(str, 'author')
...     year = TypeDesc(int, 'year')
```

Vogliamo che `Book` abbia un attributo `_order` che faccia riferimento a una lista contenente i nomi degli attributi, nell'ordine con cui questi sono definiti nell'istruzione `class`:

```
>>> Book._order
['title', 'author', 'year', 'isbn', 'publisher']
```

Scrivere un test unitario e verificare che fallisca

La soluzione più banale al nostro problema consiste nello scrivere a mano la lista:

```
>>> from typedesc import TypeDesc
>>> class Book:
...     title = TypeDesc(str, 'title')
...     author = TypeDesc(str, 'author')
...     year = TypeDesc(int, 'year')
...     _order = ['title', 'author', 'year']
...
>>> Book._order
['title', 'author', 'year']
```

Come abbiamo detto nella sezione *Ordine di definizione degli attributi*

nell'istruzione class, questa idea non è di certo geniale. Una soluzione più furba consiste nel definire una metaclassa che, all'atto della creazione della classe, crea l'attributo `_order`, facendo quindi il lavoro per noi. Il primo passo consiste, quindi, nel definire una metaclassa che non fa nulla:

```
$ $ cat orderedtypemeta.py
from typedesc import TypeDesc
class OrderedTypeMeta(type):
    pass
```

A questo punto scriviamo il test unitario:

```
$ cat test_orderedtype.py
import unittest
from orderedtypemeta import OrderedTypeMeta
from typedesc import TypeDesc
class TestBook(unittest.TestCase):
    def setUp(self):
        class Book(metaclass=OrderedTypeMeta):
            title = TypeDesc(str)
            author = TypeDesc(str)
            year = TypeDesc(int)
            self.Book = Book
        self.test_order(self)
    """Verifica che l'attributo Book._order sia una lista ordinata come da attese."""
    self.assertEqual(self.Book._order, ['title', 'author', 'year'])
if __name__ == '__main__':
    unittest.main()
```

È presente un solo test, che si occupa di verificare che la lista `Book._order` sia uguale a quella prevista. Eseguiamo il test con lo scopo di accertarci che fallisca:

```
$ python test_orderedtype.py
E
-----
ERROR: test_order (__main__.TestBook)
Verifica che l'attributo Book._order sia una lista ordinata come da attese.
-----
Traceback (most recent call last):
...
AttributeError: type object 'Book' has no attribute '_order'
-----
Ran 1 test in 0.000s
FAILED (errors=1)
```

Il test fallisce perché la classe `Book` non ha l'attributo `_order` e questo è ciò che volevamo.

Implementare le funzionalità richieste

Implementiamo la metaclassa `OrderedTypeMeta`:

```
$ cat orderedtypemeta.py
from collections import OrderedDict
from typedesc import TypeDesc
class OrderedTypeMeta(type):
    f __prepare__(clsname, bases):
        return OrderedDict(_order=[])
    f __new__(metacls, clsname, bases, namespace):
        for key, value in ((k, v) for k, v in namespace.items()):
            if isinstance(value, TypeDesc):
                value._name = key
                namespace['_order'].append(value._name)
        return super().__new__(metacls, clsname, bases, namespace)
```

Eseguiamo il test:

```
$ python test_orderedtype.py
```

```
.....
```

```
Ran 1 test in 0.000s
```

```
OK
```

Benissimo, la nostra metaclassa sembra funzionare a dovere. Prendiamoci, quindi, qualche minuto per analizzare il codice, descrivendo ciò che accade quando viene creata la classe `Book`:

```
class Book(metaclass=OrderedTypeMeta):
    title = TypeDesc(str)
    author = TypeDesc(str)
    year = TypeDesc(int)
```

Prima di tutto viene chiamato il metodo `OrderedMeta.__prepare__()`, il quale restituisce un dizionario ordinato contenente la chiave `_order`, che fa riferimento a una lista vuota. Quando `__prepare__()` ha terminato il suo lavoro, Python aggiunge al dizionario, nell'ordine, gli attributi `title`, `author` e `year`, più gli attributi magici speciali, come, ad esempio, `__qualname__`. A questo punto viene chiamato il metodo `OrderedMeta.__new__()`, che deve istanziare la classe `Book`. Come prima cosa, questo metodo itera sulle coppie `{attributo: valore}`. Per ciascun valore, verifica che questo sia istanza di `TypeDesc`, nel qual caso assegna a `value._name` il nome dell'attributo. Ad esempio, nel caso di `title` si ha `title._name = 'title'`. Come abbiamo visto nelle sezioni precedenti, il nome dell'attributo viene inserito nel messaggio di errore dell'eccezione sollevata da `TypeDesc.__set__()`.

Il nome dell'attributo, a questo punto, viene aggiunto alla lista `_order`.

L'inserimento nella lista avviene secondo l'ordine di definizione degli attributi nell'istruzione `class`, perché stiamo iterando sugli elementi del dizionario ordinato restituito da `OrderedMeta.__prepare__()` e popolato da Python prima della chiamata al costruttore `OrderedMeta.__new__()`.

Infine, osserviamo che non abbiamo istanziato `TypeDesc` passandogli il nome del descriptor, come avevamo sempre fatto sinora, poiché il nome viene assegnato in modo automatico nel metodo `OrderedMeta.__new__()`.

Prima di passare al refactoring, riassumiamo il funzionamento congiunto del descriptor `TypeDesc` con la metaclass `OrderedTypeMeta`. A tale scopo consideriamo ancora una generica classe `Book`:

```
>>> from typedesc import TypeDesc
>>> from orderedtypemeta import OrderedTypeMeta
>>> class Book(metaclass=OrderedTypeMeta):
...     title = TypeDesc(str)
...     author = TypeDesc(str)
...     year = TypeDesc(int)
...     publisher = TypeDesc(str)
...     def __init__(self, title, author, year, publisher):
...         self.title, self.author, self.year, self.publisher = title, author,\
...         year, publisher
```

Questa ha l'attributo `_order` che fa riferimento a una lista contenente gli attributi di classe di `Book`, di tipo `TypeDesc`, ordinati sulla base di come sono stati definiti nell'istruzione `class`:

```
>>> Book._order
['title', 'author', 'year', 'publisher']
```

Non possiamo assegnare a un attributo di istanza un oggetto di tipo diverso rispetto a quello atteso:

```
>>> b = Book('Programmare con Python', 'Marco Buttu', '2014', 'FAG Edizioni')
Traceback (most recent call last):
...
TypeError: L'attributo 'year' deve essere di tipo 'int'
```

A presto punto, passiamo al refactoring.

Refactoring

Anche questa volta, ci sembra che il codice sia abbastanza leggibile e che, in

questo momento, non vi siano migliorie da apportare, se non documentarlo meglio con degli esempi di codice funzionante, che possiamo copiare dalla shell interattiva. Ecco il risultato al termine di questo lavoro:

```
$ cat orderedtypemeta.py
"""Questo modulo definisce la metaclassa `OrderedTypeMeta`.
Se una generica classe `Foo` ha come metaclassa `OrderedTypeMeta`, allora `Foo`
ha un attributo `_order` che fa riferimento a una lista contenente gli attributi
di `Foo` di tipo `TypeDesc`, ordinati secondo l'ordine con cui sono definiti nella
istruzione class:
```

```
>>> class Foo(metaclass=OrderedTypeMeta):
...     c = TypeDesc(int)
...     x = TypeDesc(str)
...     a = TypeDesc(int)
...     m = TypeDesc(list)
...
>>> Foo._order
['c', 'x', 'a', 'm']
```

Inoltre, il descriptor `TypeDesc` fa sì che agli attributi di istanza possano essere assegnati solamente istanze del tipo passato a `TypeDesc`:

```
>>> f = Foo()
>>> f.c = 'python' # Errore: posso assegnare solamente degli interi
Traceback (most recent call last):
```

```
...
TypeError: L'attributo `c` deve essere di tipo `int`
"""
```

```
from collections import OrderedDict
from typedesc import TypeDesc
```

```
class OrderedTypeMeta(type):
    f __prepare__(clsname, bases):
        return OrderedDict(_order=[])
    f __new__(metaclasses, clsname, bases, namespace):
        for key, value in ((k, v) for k, v in namespace.items()):
            if isinstance(value, TypeDesc):
                value._name = key
                namespace['_order'].append(value._name)
        return super().__new__(metaclasses, clsname, bases, namespace)
```

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Anche questa volta abbiamo usato il modulo `doctest`, in modo da validare le docstring. A questo punto effettuiamo subito il docstring validation testing, eseguendo il modulo:

```
$ python orderedtypemeta.py -v
...
4 passed and 0 failed.
Test passed.
```

Benissimo, gli esempi riportati nelle docstring funzionano correttamente. Rieseguiamo anche i test unitari, in modo da accertarci di non aver introdotto degli errori durante il refactoring:

```
$ python test_orderedtype.py
.
-----
Ran 1 test in 0.000s
OK
```

Il nostro lavoro sulla classe `OrderedTypeMeta` è terminato.

A questo punto, se vogliamo approfondire l'argomento sul testing con Python, un buon punto di partenza può essere wiki.python.org/moin/PythonTestingToolsTaxonomy.

Le enumerazioni

Nel Capitolo 2 abbiamo detto che con Python 3.4 sono state introdotte le enumerazioni. È arrivato il momento di approfondire il discorso.

Per creare una enumerazione possiamo seguire due strade equivalenti. La prima consiste nell'istanziare la classe `enum.Enum`, in modo identico a quanto visto nel Capitolo 2 per `enum.IntEnum`:

```
>>> from enum import Enum
>>> Pasta = Enum('Pasta', 'spaghetti lasagne tagliatelle')
>>> Pasta.spaghetti
<Pasta.spaghetti: 1>
>>> Pasta.lasagne
<Pasta.lasagne: 2>
>>> Pasta.tagliatelle
<Pasta.tagliatelle: 3>
```

La seconda modalità consiste nel creare una classe che eredita da `enum.Enum`:

```
>>> class Pasta(Enum):
...     spaghetti = 1
...     lasagne = 2
...     tagliatelle = 3
...
>>> Pasta.spaghetti
<Pasta.spaghetti: 1>
>>> Pasta.lasagne
<Pasta.lasagne: 2>
>>> Pasta.tagliatelle
<Pasta.tagliatelle: 3>
```

I membri di una enumerazione

I membri di una enumerazione di tipo `enum.Enum`, a differenza dei membri di una enumerazione di tipo `IntEnum`, non sono degli interi:

```
>>> isinstance(Pasta.spaghetti, int)
False
```

Il valore dei membri può quindi essere un oggetto di tipo qualunque:

```
>>> class Pasta(Enum):
...     spaghetti = 'uno'
```

```
... lasagne = 'due'
... tagliatelle = [1, 2, 3]
...
>>> Pasta.tagliatelle
<Pasta.tagliatelle: [1, 2, 3]>
```

L'attributo `__members__` di una enumerazione fa riferimento a un dizionario che ha per chiavi i nomi dei membri e per valori i corrispondenti membri:

```
>>> Pasta.__members__['spaghetti'], Pasta.__members__['lasagne']
(<Pasta.spaghetti: 'uno'>, <Pasta.lasagne: 'due'>)
```

Questo dizionario è ordinato secondo l'ordine di definizione degli attributi e non è un oggetto di tipo `dict` ma di tipo `types.MappingProxyType`:

```
>>> import types
>>> isinstance(Pasta.__members__, types.MappingProxyType)
True
```

Le enumerazioni sono oggetti iterabili e, quando si itera su di essi, si itera sui membri in modo ordinato sulla base di come sono stati definiti:

```
>>> for member in Pasta:
...     print(member.name, member.value, sep=' --> ')
...
spaghetti --> uno
lasagne --> due
tagliatelle --> [1, 2, 3]
```

I membri non possono essere modificati:

```
>>> Pasta.spaghetti = 1
Traceback (most recent call last):
...
AttributeError: Cannot reassign members.
```

Sono quindi degli oggetti immutabili, pertanto possono essere utilizzati come elementi di un set:

```
>>> {member for member in Pasta}
{<Pasta.lasagne: 'due'>, <Pasta.tagliatelle: [1, 2, 3]>, <Pasta.spaghetti: 'uno'>}
```

oppure come chiavi di un dizionario:

```
>>> {member: 'wow!' for member in Pasta}
```

```
{<Pasta.lasagne: 'due'>: 'wow!', <Pasta.tagliatelle: [1, 2, 3]>: 'wow!', <Pasta.spaghetti: 'uno'>: 'wow!'}
```

Come abbiamo visto nel Capitolo 2, i membri hanno un nome e un valore, rappresentati dai loro attributi `name` e `value`:

```
>>> Pasta.spaghetti.name
'spaghetti'
>>> Pasta.spaghetti.value
'uno'
```

Noto il valore di un membro, il modo più semplice per ottenere il membro consiste nel chiamare la classe passandole il valore:

```
>>> Pasta('uno')
<Pasta.spaghetti: 'uno'>
```

Se, invece, è noto il nome di un membro e vogliamo ottenere il membro, il modo più semplice è il seguente:

```
>>> Pasta['spaghetti']
<Pasta.spaghetti: 1>
```

Questo è equivalente a chiedere il membro tramite il dizionario `Pasta.__members__`:

```
>>> Pasta['spaghetti']
<Pasta.spaghetti: 'uno'>
```

Una enumerazione non può avere due membri con lo stesso nome:

```
>>> class Pasta(Enum):
...     spaghetti = 1
...     spaghetti = 2
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'spaghetti'
```

È invece possibile che due membri abbiano il medesimo valore, ma, in questo caso, il secondo in ordine di definizione non è altro che un alias del primo:

```
>>> class Pasta(Enum):
...     spaghetti = 1
...     lasagne = 2
```

```
... tagliatelle = 1
...
>>> Pasta['tagliatelle']
<Pasta.spaghetti: 1>
```

Quando si itera su una enumerazione, gli alias non vengono considerati:

```
>>> for member in Pasta:
...     print(member)
...
Pasta.spaghetti
Pasta.lasagne
```

Se vogliamo ottenere anche gli alias, dobbiamo iterare sul dizionario `__members__`:

```
>>> for member in Pasta.__members__.values():
...     print(member)
...
Pasta.spaghetti
Pasta.lasagne
Pasta.spaghetti
```

Se non vogliamo che sia possibile creare degli alias, possiamo decorare la classe con `enum.unique`:

```
>>> from enum import unique
>>> @unique
... class Pasta(Enum):
...     spaghetti = 1
...     lasagne = 2
...     tagliatelle = 1
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Pasta'>: tagliatelle -> spaghetti
```

In una enumerazione generica, a differenza di quanto accade per le enumerazioni di interi, non possiamo fare un confronto per stabilire se un membro è più grande o più piccolo di un altro membro:

```
>>> Pasta.spaghetti < Pasta.lasagne
Traceback (most recent call last):
...
TypeError: unorderable types: Pasta() < Pasta()
```


Possiamo fare un confronto di uguaglianza e, in questo caso, i membri, diversamente da quanto accade per quelli delle enumerazioni di interi, vengono confrontati per identità, per cui il confronto con il loro valore restituisce `False`:

```
>>> Pasta.spaghetti == Pasta.tagliatelle
True
>>> Pasta.spaghetti == 1 # Se l'enumerazione fosse di interi verrebbe restituito True
False
>>> Pasta.spaghetti.value == 1
True
```

Nel caso delle enumerazioni di interi viene confrontato il valore:

```
>>> from enum import IntEnum
>>> class Pasta(IntEnum):
...     spaghetti = 1
...     lasagne = 2
...     tagliatelle = 1
...
>>> Pasta.spaghetti == 1
True
```

Chiudiamo questa sezione osservando che i membri sono istanze dell'enumerazione:

```
>>> isinstance(Pasta.spaghetti, Pasta)
True
```

Esercizio conclusivo

In questo esercizio conclusivo analizzeremo una classe che si comporta come `enum.Enum`. L'esercizio è molto significativo, poiché ci consentirà di ripassare e approfondire quanto abbiamo studiato nel capitolo. Il codice è il seguente:

```
$ cat myenum.py
"""Questo modulo consente di creare delle enumerazioni.
```

Una enumerazione è una classe che eredita da `'MyEnum'`. Ad esempio, la seguente classe `'Pasta'` è una enumerazione:

```
>>> class Pasta(MyEnum):
...     spaghetti = 1
...     lasagne = 2
...     tagliatelle = 3
```

Gli attributi di classe `'Pasta.spaghetti'`, `'Pasta.lasagne'` e `'Pasta.tagliatelle'` sono detti membri. Questi sono istanze di `'Pasta'` e hanno un nome e un valore:

```
>>> isinstance(Pasta.spaghetti, Pasta)
True
>>> Pasta.lasagne.name
'lasagne'
>>> Pasta.lasagne.value
2
```

I membri non possono essere riassegnati:

```
>>> Pasta.tagliatelle = 1
Traceback (most recent call last):
...
AttributeError: Non si possono riassegnare i membri
"""
import types
from collections import OrderedDict
```

```
class Namespace(OrderedDict):
    f __setitem__(self, name, value):
        if name in self:
            raise KeyError("Un attributo di nome '%s' esiste già" %name)
        else:
            super().__setitem__(name, value)
```

```
class MyEnumMeta(type):
```

```

f __prepare__(clsname, bases):
    return Namespace()
f __new__(metacls, clsname, bases, namespace, *args, **kwargs):
    cls = super().__new__(metacls, clsname, bases, namespace)
    members = OrderedDict()
    for name, value in namespace.items():
        if not name.startswith('__') and not name.endswith('__'):
            for member in members.values():
                if member.value == value:
                    attr = member
                    break
            else:
                attr = cls(name, value)
                setattr(cls, name, attr)
            members[name] = attr
    setattr(cls, '__members__', types.MappingProxyType(members))
    return cls
f __setattr__(self, name, value):
    if hasattr(self, name) and isinstance(getattr(self, name), self):
        raise AttributeError('Non si possono riassegnare i membri')
    else:
        super().__setattr__(name, value)
f __delattr__(self, name):
    if hasattr(self, name) and isinstance(getattr(self, name), self):
        raise AttributeError('Non si possono cancellare i membri')
    else:
        super().__delattr__(name)
f __iter__(self):
    unique_members = []
    for member in self.__members__.values():
        if not member in unique_members:
            unique_members.append(member)
    return iter(unique_members)
f __getitem__(self, name):
    for member in self:
        if member.name == name:
            return member
    raise KeyError('Un membro con questo nome non esiste')
f __call__(self, *args, **kwargs):
    if len(args) == 1:
        for member in self:
            if member.value == args[0]:
                return member
    raise ValueError('Un membro con questo valore non esiste')
    else:
        return super().__call__(*args, **kwargs)

class MyEnum(metaclass=MyEnumMeta):
    f __init__(self, name, value):
        self.name = name
        self.value = value
    f __str__(self):
        return '%s.%s' %(type(self).__name__, self.name)
    f __repr__(self):
        return '<%s.%s: %s>' %(type(self).__name__, self.name, self.value)

```

```
if __name__ == '__main__':
import doctest
doctest.testmod()
```

Probabilmente questo codice vi sembra incomprensibile. Per il momento leggetelo ancora varie volte, cercando di capirne il significato in modo autonomo. Nelle prossime sezioni chiariremo il significato di ogni metodo, utilizzando il test driven development. Per ogni funzionalità creeremo un test e solo dopo implementeremo la funzionalità. Eseguiremo il refactoring una volta che tutte le funzionalità saranno implementate.

Come al solito, troviamo il codice sorgente all'URL <http://code.google.com/p/the-pythonic-way/>.

Membri ordinati, membri istanze dell'enumerazione e membri con attributi name e value

Come abbiamo visto nella sezione sulle enumerazioni, l'attributo `__members__` fa riferimento a un dizionario ordinato le cui chiavi sono i nomi dei membri, ordinati secondo l'ordine di definizione, e i valori i corrispondenti membri. Inoltre abbiamo detto che i membri sono istanze dell'enumerazione e hanno gli attributi `name` e `value`. Queste sono le prime funzionalità che implementeremo; ciò significa che, data una classe `Pasta` che eredita da `MyEnum`:

```
>>> from myenum import MyEnum
>>> class Pasta(MyEnum):
...     spaghetti = 1
...     lasagne = 2
...     tagliatelle = 3
```

vogliamo che il dizionario `Pasta.__members__` sia ordinato secondo l'ordine di definizione dei membri nell'istruzione `class`, ovvero `'spaghetti'`, `'lasagne'` e `'tagliatelle'`:

```
>>> list(Pasta.__members__)
['spaghetti', 'lasagne', 'tagliatelle']
```

Vogliamo, inoltre, che i membri siano istanze di `Pasta`:

```
>>> isinstance(Pasta.spaghetti, Pasta)
True
>>> isinstance(Pasta.lasagne, Pasta)
True
>>> isinstance(Pasta.tagliatelle, Pasta)
True
```

Infine, vogliamo che i membri abbiano gli attributi `name` e `value`:

```
>>> Pasta.spaghetti.name
'spaghetti'
>>> Pasta.spaghetti.value
1
```

Scrivere un test unitario e verificare che fallisca

Come ormai sappiamo, il primo passo consiste nello scrivere un test unitario e verificare che fallisca. Questo è il test:

```
$ cat test_myenum.py
import unittest
from myenum import MyEnum
class TestPasta(unittest.TestCase):
    def setUp(self):
        class Pasta(MyEnum):
            spaghetti = 1
            lasagne = 2
            tagliatelle = 3
        self.Pasta = Pasta
    def test_membersOrder(self):
        """Verifica che i membri siano ordinati secondo l'ordine di definizione."""
        self.assertEqual(['spaghetti', 'lasagne', 'tagliatelle'],
            list(self.Pasta.__members__))
    def test_isInstance(self):
        """Verifica che i membri siano istanze della classe Pasta."""
        for member in self.Pasta.__members__.values():
            self.assertIsInstance(member, self.Pasta)
    def test_memberAttributes(self):
        """Verifica che gli attributi name e value dei membri siano corretti."""
        self.assertEqual(self.Pasta.spaghetti.name, 'spaghetti')
        self.assertEqual(self.Pasta.spaghetti.value, 1)

if __name__ == '__main__':
    unittest.main()
```

Come possiamo vedere, nel `setUp()` viene creata una enumerazione, che poi viene utilizzata nei test. I tre test sono abbastanza autoesplicativi, per cui non ci dilungheremo nella loro spiegazione.

Creiamo, quindi, il modulo `myenum` e scriviamo una classe `MyEnum` che non fa nulla, in modo da verificare che il test fallisca:

```
$ cat myenum.py
class MyEnum:
    pass
```

Eseguiamo il test:

```
$ python test_myenum.py
EEE
```

```
...
-----
Ran 3 tests in 0.000s
```

FAILED (errors=3)

Il test è fallito, per cui possiamo andare avanti e procedere con l'implementazione.

Implementare le funzionalità richieste

Implementiamo le funzionalità:

```
$ cat myenum.py
import types
from collections import OrderedDict
class MyEnumMeta(type):
    f __prepare__(clsname, bases):
    return OrderedDict()
    f __new__(metacls, clsname, bases, namespace, *args, **kwargs):
    cls = super().__new__(metacls, clsname, bases, namespace)
    members = OrderedDict()
    for name, value in namespace.items():
    if not name.startswith('__') and not name.endswith('__'):
    attr = cls(name, value)
    setattr(cls, name, attr)
    members[name] = attr
    setattr(cls, '__members__', types.MappingProxyType(members))
    return cls
```

```
class MyEnum(metaclass=MyEnumMeta):
    f __init__(self, name, value):
    self.name = name
    self.value = value
```

Tra breve analizzeremo il codice, ma prima per fare ciò verifichiamo che il test venga superato:

```
$ python test_myenum.py
```

```
...
```

```
-----
Ran 3 tests in 0.000s
```

```
OK
```

Bene, abbiamo superato il test, per cui siamo pronti ad analizzare in dettaglio il codice, linea per linea.

La classe `MyEnum` ha metaclasses `MyEnumMeta`, per cui anche `Pasta`, visto che eredita

da `MyEnum`, ha metaclass `MyEnumMeta`. Il dizionario a cui fa riferimento l'attributo `Pasta.__members__` è ordinato secondo l'ordine di definizione dei membri, poiché, come vedremo tra breve, è creato iterando sul dizionario restituito da `MyEnumMeta.__prepare__()`, il quale è ordinato.

Il metodo `MyEnumMeta.__new__()` crea i membri in modo che siano istanze di `Pasta`. Prima, infatti, viene creata la classe `Pasta` (istanza di `MyEnumMeta`) assegnata all'etichetta `cls`:

```
cls = super().__new__(metaclass, clsname, bases, namespace)
```

Viene poi creato il dizionario ordinato `members`:

```
members = OrderedDict()
```

A questo punto si itera sul dizionario restituito da `MyEnumMeta.__prepare__()`, contenente gli attributi di `Pasta`, ordinati secondo l'ordine di definizione:

```
for name, value in namespace.items():
    if not name.startswith('__') and not name.endswith('__'):
        attr = cls(name, value)
        setattr(cls, name, attr)
        members[name] = attr
setattr(cls, '__members__', types.MappingProxyType(members))
return cls
```

Come possiamo vedere, per ogni attributo viene verificato che non sia un attributo speciale, controllando che il suo nome non inizi e non finisca con un doppio underscore. Se non è speciale, allora viene creato un membro, istanziando `Pasta`. Ad esempio, per il primo attributo, `Pasta.spaghetti`, il nome `name` è `'spaghetti'` e il valore `value` è `1`, per cui `attr = cls(name, value)` è `Pasta('spaghetti', 1)`. Il metodo `Pasta.__init__()` è `MyEnum.__init__()`, visto che `Pasta` non fa l'overriding di quest'ultimo. Quindi, la chiamata `Pasta('spaghetti', 1)` durante l'inizializzazione assegna (all'istanza) `self.name = 'spaghetti'` e `self.value = 1`. Il risultato di tutto ciò è che `attr` è una istanza di `Pasta`, tale che `attr.name` fa riferimento alla stringa `'spaghetti'`, mentre `attr.value` fa riferimento a `1`. Questo è ciò che era richiesto dalla seconda e dalla terza funzionalità.

A questo punto, creato il membro, dobbiamo assegnarlo alla classe `Pasta`, in modo che `Pasta.spaghetti` faccia riferimento a esso. Per fare ciò usiamo `setattr()`, perché non possiamo fare l'assegnamento `Pasta.__dict__['spaghetti'] = attr`, visto che il namespace di classe non è implementato con un dizionario ordinario, ma è un oggetto di tipo `types.MappingProxyType` (vedi sezione *Differenze di implementazione tra il namespace di classe e di istanza* del [Capitolo 5](#)).

Fatto ciò, il membro viene aggiunto al dizionario ordinato `members`. Queste operazioni vengono compiute per ogni membro non speciale, per cui, alla fine dell'iterazione su `namesapece`, il dizionario `members` contiene tre elementi, ciascuno dei quali ha per chiave il nome del membro e per valore il membro stesso. A questo punto `members` viene assegnato all'attributo `Pasta.__members__`, ancora una volta tramite `setattr()`. Nella sezione *I membri di una enumerazione* abbiamo visto come `__members__` faccia riferimento a un oggetto di tipo `types.MappingProxyType` e questo è il motivo per cui non abbiamo assegnato direttamente:

```
setattr(cls, '__members__', members)
```

ma piuttosto:

```
setattr(cls, '__members__', types.MappingProxyType(members))
```

Se non ci sono rimasti dubbi su questa parte, allora il resto dell'esercizio è in discesa e non faremo alcuna fatica a comprenderlo. In caso contrario, è bene rileggere con attenzione questa sezione.

Non vi possono essere due membri con lo stesso nome

Come sappiamo, se creiamo due attributi con lo stesso nome, l'ultimo assegnamento è quello utile:

```
>>> class Foo:
...     a = 33
...     a = 44
...
>>> Foo.a
44
```

Nel caso delle enumerazioni, abbiamo visto che questo non può accadere, perché, se vengono definiti due membri con lo stesso nome, viene sollevata una eccezione:

```
>>> import enum
>>> class Foo(enum.Enum):
...     a = 33
...     a = 44
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'a'
```


Vogliamo che `MyEnum` si comporti allo stesso modo di `enum.Enum`, per cui implementiamo questa funzionalità.

Scrivere un test unitario e verificare che fallisca

Aggiungiamo al precedente test case il seguente test:

```
def test_noHomonym(self):
    """Verifica che non vi siano membri con lo stesso nome."""
    namespace = Namespace({'spaghetti': 1})
    self.assertRaises(KeyError, namespace.update, {'spaghetti': 1})
```

Implementeremo la classe `Namespace` in *myenum.py*. Erediterà da un dizionario ordinato e farà in modo che, se una chiave è già presente, venga sollevata una eccezione di tipo `KeyError`. Scriviamo questa classe, senza implementarla, in *myenum.py*, nel modo seguente:

```
class Namespace(OrderedDict):
    pass
```

Eseguiamo il test per verificare che fallisca:

```
$ python test_myenum.py
...F
```

```
FAIL: test_noHomonym (__main__.TestPasta)
Verifica che non vi siano membri con lo stesso nome.
```

```
-----
Traceback (most recent call last):
```

```
...
AssertionError: KeyError not raised by update
```

```
-----
Ran 4 tests in 0.001s
```

```
FAILED (failures=1)
```

Bene, il test è fallito per cui possiamo procedere con l'implementazione della classe `Namespace`.

Implementare la funzionalità richiesta

Implementiamo il namespace nel modo seguente:

```
class Namespace(OrderedDict):
    f __setitem__(self, name, value):
        if name in self:
            raise KeyError("Un attributo di nome '%s' esiste già" % name)
```

```
else:
    super().__setitem__(name, value)
```

Il suo significato dovrebbe essere chiaro, per cui non ci soffermeremo per ulteriori spiegazioni. Eseguiamo quindi il test:

```
$ python test_myenum.py
```

```
....
```

```
-----
```

```
Ran 4 tests in 0.001s
```

```
OK
```

Il test è stato superato, per cui passiamo alla prossima funzionalità.

I membri non possono essere né riassegnati né cancellati

Vogliamo che i membri non possano essere né riassegnati né cancellati, così come accade per le enumerazioni che ereditano da `enum.Enum`:

```
>>> class Pasta(enum.Enum):
```

```
...     spaghetti = 1
```

```
...     lasagne = 2
```

```
...     tagliatelle = 3
```

```
...
```

```
>>> Pasta.spaghetti = 2
```

```
Traceback (most recent call last):
```

```
...
```

```
AttributeError: Cannot reassign members.
```

```
>>> del Pasta.spaghetti
```

```
Traceback (most recent call last):
```

```
...
```

```
AttributeError: Pasta: cannot delete Enum member.
```

Scrivere un test unitario e verificare che fallisca

Aggiungiamo al precedente test case il seguente test:

```
def test_doNotChange(self):
```

```
    """Verifica che i membri non possano essere né riassegnati né cancellati."""
```

```
    self.assertRaises(AttributeError, setattr, self.Pasta, 'spaghetti', 2)
```

```
    self.assertRaises(AttributeError, delattr, self.Pasta, 'spaghetti')
```

Eseguiamo il test e verifichiamo che fallisca:

```
$ python test_myenum.py
```

```
F....
```

```
FAIL: test_doNotChange (__main__.TestPasta)
```

Verifica che i membri non possano essere né riassegnati né cancellati.

```
-----  
Traceback (most recent call last):
```

```
...
```

```
AssertionError: AttributeError not raised by setattr
```

```
-----  
Ran 5 tests in 0.001s
```

```
FAILED (failures=1)
```

Il test è fallito, per cui possiamo procedere con l'implementazione della funzionalità.

Implementare la funzionalità richiesta

Come probabilmente avremo intuito, dobbiamo definire i metodi `__setattr__()` e `__delattr__()`. Aggiungiamoli, quindi, alla nostra classe `MyEnumMeta`, definendoli nel modo seguente:

```
def __setattr__(self, name, value):  
    if hasattr(self, name) and isinstance(getattr(self, name), self):  
        raise AttributeError('Non si possono riassegnare i membri')  
    else:  
        super().__setattr__(name, value)  
def __delattr__(self, name):  
    if hasattr(self, name) and isinstance(getattr(self, name), self):  
        raise AttributeError('Non si possono cancellare i membri')  
    else:  
        super().__delattr__(name)
```

Eseguiamo il test:

```
$ python test_myenum.py
```

```
.....
```

```
-----  
Ran 5 tests in 0.001s
```

```
OK
```

Possiamo passare alla prossima funzionalità.

Un membro con lo stesso valore di uno esistente è un alias

Se un membro ha lo stesso valore di un altro, vorremmo che fosse un alias di quest'ultimo, come accade per le enumerazioni derivate da `enum.Enum`:

```
>>> class Pasta(enum.Enum):
```

```
...     spaghetti = 1
```

```
...     lasagne = 2
```

```
...     tagliatelle = 1
```

```
...
>>> Pasta.tagliatelle
<Pasta.spaghetti: 1>
```

Scrivere un test unitario e verificare che fallisca

Aggiungiamo al nostro test case il seguente test:

```
def test_aliases(self):
    """Verifica che un membro con lo stesso valore di uno esistente sia un alias."""
    class Pasta(MyEnum):
        spaghetti = 1
        lasagne = 2
        tagliatelle = 1
    self.assertIs(Pasta.spaghetti, Pasta.tagliatelle)
```

Verifichiamo che fallisca:

```
$ python test_myenum.py
F.....
```

```
FAIL: test_aliases (__main__.TestPasta)
Verifica che un membro con lo stesso valore di uno esistente sia un alias.
-----
Traceback (most recent call last):
...
AssertionError: <__main__.TestPasta.test_aliases.<locals>.Pasta object ...>
-----
Ran 6 tests in 0.002s
```

FAILED (failures=1)

Possiamo procedere con l'implementazione della funzionalità richiesta.

Implementare la funzionalità richiesta

Riscriviamo il metodo `MyEnumMeta.__new__()` come mostrato di seguito:

```
def __new__(metacls, clsname, bases, namespace, *args, **kwargs):
    cls = super().__new__(metacls, clsname, bases, namespace)
    members = OrderedDict()
    for name, value in namespace.items():
        if not name.startswith('__') and not name.endswith('__'):
            for member in members.values():
                if member.value == value:
                    attr = member
                    break
            else:
                attr = cls(name, value)
            setattr(cls, name, attr)
            members[name] = attr
    setattr(cls, '__members__', types.MappingProxyType(members))
    return cls
```

In questo modo, prima che il membro venga creato, viene verificato che non vi sia già un membro il cui valore è pari a quello che gli si vorrebbe assegnare. Se un membro con quel valore esiste già, allora viene creato un alias a tale membro esistente, altrimenti viene creato un nuovo membro. Eseguiamo il test:

```
$ python test_myenum.py
```

```
.....
```

```
-----  
Ran 6 tests in 0.002s
```

OK

Possiamo passare alla prossima funzionalità.

Le enumerazioni sono oggetti iterabili

Vogliamo che le nostre enumerazioni siano oggetti iterabili, così come lo sono quelle che ereditano da `enum.Enum`:

```
>>> class Pasta(enum.Enum):
```

```
...     spaghetti = 1
```

```
...     lasagne = 2
```

```
...     tagliatelle = 3
```

```
...
```

```
>>> for member in Pasta:
```

```
...     print(member)
```

```
...
```

```
Pasta.spaghetti
```

```
Pasta.lasagne
```

```
Pasta.tagliatelle
```

Come possiamo vedere, queste iterano sui membri. Vogliamo, inoltre, che nelle iterazioni gli alias non vengano considerati:

```
>>> class Pasta(enum.Enum):
```

```
...     spaghetti = 1
```

```
...     lasagne = 2
```

```
...     tagliatelle = 1
```

```
...
```

```
>>> for member in Pasta:
```

```
...     print(member)
```

```
...
```

```
Pasta.spaghetti
```

```
Pasta.lasagne
```

Scrivere i test unitari e verificare che falliscano

Aggiungiamo i seguenti test al nostro test case:

```
def test_iterable(self):
    """Verifica che le enumerazioni siano oggetti iterabili."""
    self.assertEqual(self.Pasta.__members__.values(), list(self.Pasta))
def test_aliasAndIterations(self):
    """Verifica che gli alias non compaiano quando si itera sulla enumerazione."""
    desired = [self.PastaAlias.spaghetti, self.PastaAlias.lasagne]
    self.assertEqual(desired, list(self.PastaAlias))
```

Definiamo la classe `self.PastaAlias` nella test fixture, come mostrato di seguito:

```
def setUp(self):
    class Pasta(MyEnum):
        spaghetti = 1
        lasagne = 2
        tagliatelle = 3
    self.Pasta = Pasta
    class PastaAlias(MyEnum):
        spaghetti = 1
        lasagne = 2
        tagliatelle = 1
    self.PastaAlias = PastaAlias
```

Verifichiamo che i test falliscano:

```
$ python test_myenum.py
E...E...
```

```
ERROR: test_aliasAndIterations (__main__.TestPasta)
Verifica che gli alias non compaiano quando si itera sulla enumerazione.
```

```
-----
Traceback (most recent call last):
```

```
...
TypeError: 'MyEnumMeta' object is not iterable
```

```
ERROR: test_iterable (__main__.TestPasta)
Verifica che le enumerazioni siano oggetti iterabili.
```

```
-----
Traceback (most recent call last):
```

```
...
TypeError: 'MyEnumMeta' object is not iterable
```

```
-----
Ran 8 tests in 0.002s
FAILED (errors=2)
```

Possiamo implementare le funzionalità.

Implementare le funzionalità richieste

Aggiungiamo il metodo `__iter__()` alla classe `MyEnumMeta`:

```
def __iter__(self):
    unique_members = []
    for member in self.__members__.values():
        if not member in unique_members:
            unique_members.append(member)
    return iter(unique_members)
```

Il metodo restituisce un iteratore che itera sugli elementi della lista `unique_members`, la quale non contiene gli alias.

Eseguiamo il test:

```
$ python test_myenum.py
```

```
.....
```

```
-----
Ran 8 tests in 0.002s
```

```
OK
```

Possiamo passare alla prossima funzionalità.

Pasta[‘nome_membro’] deve restituire il membro

Vogliamo implementare la seguente funzionalità:

```
>>> class Pasta(enum.Enum):
```

```
...     spaghetti = 1
```

```
...     lasagne = 2
```

```
...     tagliatelle = 1
```

```
...
```

```
>>> Pasta['spaghetti']
```

```
<Pasta.spaghetti: 1>
```

Scrivere un test unitario e verificare che fallisca

Aggiungiamo al test case il seguente test:

```
def test_getitem(self):
```

```
    """Verifica che Pasta['nome_membro'] restituisca il membro."""
```

```
    self.assertIs(self.Pasta.spaghetti, self.Pasta['spaghetti'])
```

Verifichiamo che fallisca:

```
$ python test_myenum.py
```

```
...E.....
```

```
ERROR: test_getitem (__main__.TestPasta)
```

Verifica che Pasta['nome_membro'] restituisca il membro.

Traceback (most recent call last):

...

TypeError: 'MyEnumMeta' object is not subscriptable

Ran 9 tests in 0.003s

FAILED (errors=1)

Implementare la funzionalità richiesta

Come avremo intuito, dobbiamo aggiungere il metodo `MyEnumMeta.__getitem__()`:

```
def __getitem__(self, name):
    for member in self:
        if member.name == name:
            return member
    raise KeyError('Un membro con questo nome non esiste')
```

Eseguiamo il test:

```
$ python test_myenum.py
```

.....

Ran 9 tests in 0.002s

OK

Pasta(value) deve restituire il membro che ha per valore value

Vogliamo implementare la seguente funzionalità:

```
>>> class Pasta(enum.Enum):
```

```
...     spaghetti = 1
```

```
...     lasagne = 2
```

```
...     tagliatelle = 1
```

```
...
```

```
>>> Pasta(1)
```

```
<Pasta.spaghetti: 1>
```

Scrivere un test unitario e verificare che fallisca

Aggiungiamo al test case il seguente test:

```
def test_call(self):
```

```
    """Verifica che Pasta(value) restituisca il membro che ha per valore `value`."""
```

```
    self.assertEqual(self.Pasta(1), self.Pasta.spaghetti)
```

```
    self.assertEqual(self.Pasta(2), self.Pasta.lasagne)
```


Eseguiamolo per verificare che fallisca:

```
$ python test_myenum.py
..E.....
```

```
ERROR: test_call (__main__.TestPasta)
Verifica che Pasta(value) restituisca il membro che ha per valore `value`.
```

```
-----
Traceback (most recent call last):
```

```
...
TypeError: __init__() missing 1 required positional argument: 'value'
```

```
-----
Ran 10 tests in 0.003s
```

```
FAILED (errors=1)
```

Implementare la funzionalità richiesta

Come avremo intuito, dobbiamo aggiungere il metodo `MyEnumMeta.__call__()`:

```
def __call__(self, *args, **kwargs):
    if len(args) == 1:
        for member in self:
            if member.value == args[0]:
                return member
        raise ValueError('Un membro con questo valore non esiste')
    else:
        return super().__call__(*args, **kwargs)
```

Eseguiamo il test:

```
$ python test_myenum.py
```

```
.....
```

```
-----
Ran 10 tests in 0.003s
```

```
OK
```

Abbiamo implementato tutte le funzionalità che ci interessano e questo è il test case finale:

```
$ cat test_myenum.py
import unittest
from myenum import *
```

```
class TestPasta(unittest.TestCase):
    def setUp(self):
        class Pasta(MyEnum):
            spaghetti = 1
            lasagne = 2
```

```

tagliatelle = 3
self.Pasta = Pasta
class PastaAlias(MyEnum):
    spaghetti = 1
    lasagne = 2
    tagliatelle = 1
self.PastaAlias = PastaAlias
f test_membersOrder(self):
    """Verifica che i membri siano ordinati secondo l'ordine di definizione."""
    self.assertEqual(['spaghetti', 'lasagne', 'tagliatelle'], list(self.
Pasta.__members__))
f test_isInstance(self):
    """Verifica che i membri siano istanze della classe Pasta."""
    for member in self.Pasta.__members__.values():
        self.assertIsInstance(member, self.Pasta)
f test_memberAttributes(self):
    """Verifica che gli attributi name e value dei membri siano corretti."""
    self.assertEqual(self.Pasta.spaghetti.name, 'spaghetti')
    self.assertEqual(self.Pasta.spaghetti.value, 1)
f test_noHomonym(self):
    """Verifica che non vi siano membri con lo stesso nome."""
    namespace = Namespace({'spaghetti': 1})
    self.assertRaises(KeyError, namespace.update, {'spaghetti': 1})
f test_doNotChange(self):
    """Verifica che i membri non possano essere né riassegnati né cancellati."""
    self.assertRaises(AttributeError, setattr, self.Pasta, 'spaghetti', 2)
    self.assertRaises(AttributeError, delattr, self.Pasta, 'spaghetti')
f test_aliases(self):
    """Verifica che un membro con lo stesso valore di uno esistente sia un alias."""
    self.assertIs(self.PastaAlias.spaghetti, self.PastaAlias.tagliatelle)
f test_iterable(self):
    """Verifica che le enumerazioni siano oggetti iterabili."""
    self.assertEqual(self.Pasta.__members__.values(), list(self.Pasta))
f test_aliasAndIterations(self):
    """Verifica che gli alias non compaiano quando si itera sulla enumerazione."""
    desired = [self.PastaAlias.spaghetti, self.PastaAlias.lasagne]
    self.assertEqual(desired, list(self.PastaAlias))
f test_getitem(self):
    """Verifica che Pasta['nome_membro'] restituisca il membro."""
    self.assertIs(self.Pasta.spaghetti, self.Pasta['spaghetti'])
f test_call(self):
    """Verifica che Pasta(value) restituisca il membro che ha per valore `value`."""
    self.assertEqual(self.Pasta(1), self.Pasta.spaghetti)
    self.assertEqual(self.Pasta(2), self.Pasta.lasagne)
if __name__ == '__main__':
    unittest.main()

```

Siamo pronti per il refactoring.

Refactoring

Abbiamo visto come i membri delle enumerazioni che derivano da `enum.Enum` vengono rappresentati e stampati nel modo seguente:

```
>>> class Pasta(enum.Enum):
...     spaghetti = 1
...     lasagne = 2
...     tagliatelle = 1
...
>>> Pasta.spaghetti
<Pasta.spaghetti: 1>
>>> print(Pasta.spaghetti)
Pasta.spaghetti
```

Le nostre enumerazioni non hanno questo comportamento, perché ancora non abbiamo definito i metodi `MyEnum.__repr__()` e `MyEnum.__str__()`. Il risultato è che i membri sono rappresentati e stampati nel modo seguente:

```
>>> class Pasta(MyEnum):
...     spaghetti = 1
...     lasagne = 2
...     tagliatelle = 3
...
>>> Pasta.spaghetti
<__main__.Pasta object at 0x7f906a490050>
>>> print(Pasta.spaghetti)
<__main__.Pasta object at 0x7f906a490050>
```

Aggiungiamo, quindi, questi due metodi a `MyEnum`:

```
class MyEnum(metaclass=MyEnumMeta):!
    f __init__(self, name, value):
        self.name = name
        self.value = value
    f __str__(self):
        return '%s.%s' %(type(self).__name__, self.name)
    f __repr__(self):
        return '<%s.%s: %s>' %(type(self).__name__, self.name, self.value)
```

Non ci resta che aggiungere le docstring e il lavoro è concluso. Il risultato è il file *myenum.py* riportato all'inizio di questa sezione.

Appendice A

Descrizione dei comandi Unix-like utilizzati nel libro

In questo capitolo vedremo il **significato** dei vari **comandi Unix**, **variabili d'ambiente** e **metacaratteri** utilizzati nel libro. I comandi sono elencati in ordine alfabetico, mentre le sezioni dedicate ai metacaratteri e alle variabili d'ambiente si trovano al termine dell'elenco dei comandi.

cat

Il comando `cat` (abbreviazione dell'inglese *catenate*) è un comando che legge i file che gli sono specificati come parametri (o, più in generale, lo standard input) e produce sullo standard output la concatenazione del loro contenuto:

```
$ echo "fooooo file" > foo
$ echo "mooooo file" > moo
$ cat foo moo
fooooo file
mooooo file
$ cat foo
fooooo file
```

chmod

Il comando `chmod` (abbreviazione di *change mode*, “cambia modalità”) modifica i permessi di file e directory. Consideriamo, ad esempio, il seguente file:

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

Se passiamo a `chmod` l'argomento `-r` (`r` sta per *reading*), allora il proprietario del file non avrà più i permessi in lettura:

```
$ chmod -r myfile.py
$ more myfile.py # Non posso più leggerlo...
myfile.py: Permission denied
```

Il simbolo `-` viene infatti utilizzato per levare un permesso. Il simbolo `+` viene invece utilizzato per dare un permesso, per cui per ridare a *myfile.py* i permessi in lettura passiamo a `chmod` l'argomento `+r`:

```
$ chmod +r myfile.py
$ more myfile.py # Adesso possiamo nuovamente leggere il file
#!/usr/bin/env python
print('www.python.org')
```

Se vogliamo che il proprietario del file abbia i permessi di esecuzione, allora passiamo a `chmod` l'argomento `+x`:

```
$ 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:

```
$ ~/temp/myfile.py # Percorso assoluto
www.python.org
$ ../temp/myfile.py # Percorso relativo
www.python.org
```

Se il file si trova nella directory corrente e questa non fa parte dei percorsi di ricerca, non possiamo eseguirlo direttamente da linea di comando:

```
$ echo myfile.py # Il file si trova nella directory corrente
myfile.py
$ myfile.py
myfile.py: command not found
```

Infatti, se non indichiamo un percorso assoluto o relativo, il file viene cercato nei percorsi di ricerca, indicati dalla variabile d'ambiente `PATH`. Quindi, se il file si trova nella directory corrente e contiene lo shebang, il modo più semplice per eseguirlo è indicare il percorso relativo:

```
$ ./myfile.py # Percorso relativo alla directory corrente
www.python.org
```

cut

Il comando `cut` (dall'inglese *cut*, “taglia”) legge uno o più file di testo (o lo standard input), estraendo da ogni linea delle sezioni, le quali sono poi mostrate sullo standard output. Consideriamo, ad esempio, le prime cinque linee del file */etc/passwd*:

```
$ head -n 5 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
```

Come possiamo vedere, ogni linea è composta da elementi separati dal carattere di due punti. Se vogliamo visualizzare solamente il primo elemento di ogni linea, possiamo mandare l'output di `head` in ingresso a `cut`, nel modo seguente:

```
$ head -n 5 /etc/passwd | cut -d':' -f1
root
daemon
bin
sys
sync
```

Il parametro `-d` di `cut` indica il delimitatore, mentre il parametro `-f` indica l'indice dell'elemento. Ad esempio, nel caso seguente viene mostrato il quinto elemento:

```
$ head -n 5 /etc/passwd | cut -d':' -f5
/root
/usr/sbin
/bin
/dev
/bin
```

Vediamo un altro esempio. Consideriamo l'output del seguente comando:

```
$ ls -l --time-style=long-iso
total 24
-rwxr-xr-x 1 marco marco 13037 2012-06-06 14:03 csrf_migration_helper.py
-rwxr-xr-x 1 marco marco 2280 2013-03-28 21:07 django_bash_completion
```



```
-rw-r--r-- 1 marco marco 77 2012-06-06 10:47 README.TXT
```

Supponiamo di voler visualizzare solamente la data, la quale si trova in sesta posizione:

```
$ ls -l --time-style=long-iso | cut -d' ' -f6
```

```
2012-06-06
2280
```

Come possiamo vedere, il risultato non è quello atteso. Infatti, se osserviamo con maggiore attenzione l'output del comando `ls`, notiamo che nella prima riga vi sono solamente due elementi, mentre nella quarta e nella quinta il quarto elemento è separato dal quinto da un numero di spazi superiore a uno. Dobbiamo fare in modo che gli elementi siano separati da un solo spazio. Per fare ciò possiamo usare il comando `tr`, in modo da rimpiazzare ogni sequenza composta da più spazi con un singolo spazio. Per fare ciò si utilizza l'opzione `-s` di `tr`, indicando lo spazio come carattere da non replicare:

```
$ ls -l --time-style=long-iso | tr -s ' '
total 24
-rwxr-xr-x 1 marco marco 13037 2012-06-06 14:03 csrf_migration_helper.py
-rwxr-xr-x 1 marco marco 2280 2013-03-28 21:07 django_bash_completion
-rw-r--r-- 1 marco marco 77 2012-06-06 10:47 README.TXT
```

Ecco, quindi, la soluzione al nostro problema:

```
$ ls -l --time-style=long-iso | tr -s ' ' | cut -d' ' -f6
```

```
2012-06-06
2013-03-28
2012-06-06
```

Se vogliamo visualizzare, oltre alla data, anche l'ora (settimo elemento), dobbiamo passare allo switch `-f` anche l'argomento 7:

```
$ ls -l --time-style=long-iso | tr -s ' ' | cut -d' ' -f 6,7
```

```
2012-06-06 14:03
2013-03-28 21:07
2012-06-06 10:47
```

diff

Il comando `diff` ci consente di effettuare un confronto linea per linea tra due file di testo in modo da vedere le differenze. Consideriamo, ad esempio, questi due file:

```
$ more file_A
prima
seconda
$ more file_B
Prima
seconda
```

Come possiamo vedere, l'unica differenza sta nella prima linea (iniziale minuscola in *file_A* e maiuscola in *file_B*). Ecco l'output del comando `diff`:

```
$ diff file_A file_B
1c1
< prima
---
> Prima
```

Questo, in sostanza, mostra le informazioni necessarie per poter rendere i due file uguali. Nella prima linea, infatti, è riportato `1c1`, e significa che la linea numero 1 di *file_A* va cambiata (il carattere `c` sta per *change*) con la linea numero 1 di *file_B*. Vengono mostrate anche le due linee: quella del primo file è preceduta dal carattere `<`, mentre quella del secondo file dal carattere `>`.

Se modifichiamo la prima linea del *file_A* rendendola uguale a quella del *file_B*, `diff` non troverà differenze tra i due file e quindi non mostrerà alcun output:

```
$ sed -i 's/prima/Prima/g' file_A # Sostituiamo nel file_A la parola `prima` con `Prima`
$ diff file_A file_B # Non vi sono differenze
$
```

Nel caso di file binari, non vengono mostrate le differenze, ma viene riportato esclusivamente se i due file differiscono oppure no. Consideriamo, ad esempio, i seguenti due file binari:

```
$ ls -l
total 8
```

```
-rw-r--r-- 1 marco marco 177 May 22 20:43 m.cpython-33.pyc  
-rw-r--r-- 1 marco marco 136 May 22 20:44 m.cpython-33.pyo
```

Come possiamo vedere, questi differiscono (uno è di 177 byte e l'altro di 136 byte), ma `diff` non mostra le differenze:

```
$ diff m.cpython-33.pyc m.cpython-33.pyo  
Binary files m.cpython-33.pyc and m.cpython-33.pyo differ
```

Nel caso in cui i file binari siano identici, allo stesso modo dei file di testo identici, non viene mostrato alcun output:

```
$ cp m.cpython-33.pyc foo  
$ diff m.cpython-33.pyc foo  
$
```

echo

Il comando `echo` scrive i suoi parametri sullo standard output, tipicamente sul terminale da cui il comando è stato eseguito:

```
$ echo "The Pythonic Way"
The Pythonic Way
```

È usato solitamente negli script di shell per visualizzare messaggi informativi:

```
$ more myscript.sh
cd ~
echo "Ci troviamo nella directory $PWD"
$ bash myscript.sh
Ci troviamo nella directory /home/marco
```

Viene spesso utilizzato per scrivere del testo in un file, in combinazione con il metacarattere di redirezione dell'output, rappresentato dal carattere `>`. Ad esempio, nel seguente caso il testo *The Pythonic Way* viene scritto sul file *myfile*:

```
$ echo "The Pythonic Way" > myfile
```

Ecco, infatti, il contenuto del file:

```
$ more myfile # Mostra il contenuto del file
The Pythonic Way
```

Se si utilizza l'opzione `-e` allora `echo` interpreta i caratteri di escape:

```
$ echo "a\nb\nc"
a\nb\nc
$ echo -e "a\nb\nc"
a
b
c
```

find

Il comando `find` ricerca file e directory nel file system:

```
$ find /usr/local/lib/python3.3/ -name collections.py
/usr/local/lib/python3.3/dist-packages/matplotlib/collections.py
/usr/local/lib/python3.3/site-packages/matplotlib/collections.py
```

In questo caso è stato cercato il file *collections.py* a partire dalla directory */usr/local/lib/python3.3*. Lo switch `-name` supporta i metacaratteri:

```
$ find /usr/local/lib/python3.3/ -name "collect*"
/usr/local/lib/python3.3/dist-packages/matplotlib/collections.py
/usr/local/lib/python3.3/dist-packages/matplotlib/__pycache__/collections.cpython-33.pyo
/usr/local/lib/python3.3/dist-packages/matplotlib/__pycache__/collections.cpython-33.pyc
/usr/local/lib/python3.3/collections
/usr/local/lib/python3.3/site-packages/matplotlib/collections.py
/usr/local/lib/python3.3/site-packages/matplotlib/__pycache__/collections.cpython-33.pyo
/usr/local/lib/python3.3/site-packages/matplotlib/__pycache__/collections.cpython-33.pyc
```

Il supporto per le espressioni regolari è fornito dallo switch `-regex`.

grep

Il comando `grep` (*general regular expression print*) ricerca in uno o più file di testo le linee che corrispondono a uno o più modelli specificati con espressioni regolari o stringhe letterali e produce un elenco delle linee (o anche dei soli nomi di file) per cui è stata trovata corrispondenza.

Consideriamo il seguente file *pyhistory*:

```
$ more pyhistory
```

```
A. HISTORY OF THE SOFTWARE
```

```
Python was created in the early 1990s by Guido van Rossum at Stichting  
Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands  
as a successor of a language called ABC. Guido remains Python's  
principal author, although it includes many contributions from others.
```

```
In 1995, Guido continued his work on Python at the Corporation for  
National Research Initiatives (CNRI, see http://www.cnri.reston.va.us)  
in Reston, Virginia where he released several versions of the  
software.
```

Possiamo usare il comando `grep` per cercare il testo `see` all'interno del file:

```
$ grep see pyhistory
```

```
Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands  
National Research Initiatives (CNRI, see http://www.cnri.reston.va.us)
```

Come possiamo vedere, sono state visualizzate le due linee del file contenenti la parola cercata. Possiamo passare a `grep` anche delle espressioni regolari:

```
$ grep "se[a-e]" pyhistory
```

```
Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands  
National Research Initiatives (CNRI, see http://www.cnri.reston.va.us)  
in Reston, Virginia where he released several versions of the
```

Con lo switch `-A` (`--after-context`) possiamo indicare il numero di linee del file che seguono quella che verifica il pattern. Ad esempio, consideriamo il seguente caso:

```
$ grep "=" pyhistory
```

È stata mostrata la linea contenente il carattere =. Ecco come visualizzare anche le due linee successive:

```
$ grep "=" pyhistory -A 2
```

```
Python was created in the early 1990s by Guido van Rossum at Stichting  
Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands
```

Per visualizzare le linee prima di quella che verifica il pattern, si usa lo switch -B (--before-context):

```
$ grep "=" pyhistory -A 2 -B 1  
A. HISTORY OF THE SOFTWARE
```

```
Python was created in the early 1990s by Guido van Rossum at Stichting  
Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands
```

Con l'opzione -R viene fatta una ricerca ricorsiva all'interno di una directory:

```
$ grep "write" venv -R  
Binary file venv/__pycache__/__init__.cpython-33.pyc matches  
Binary file venv/__pycache__/__init__.cpython-33.pyo matches  
venv/__init__.py: f.write('home = %s\n' % context.python_dir)  
venv/__init__.py: f.write('include-system-site-packages = %s\n' % incl)  
venv/__init__.py: f.write('version = %d.%d.%d\n' % sys.version_info[:3])  
venv/__init__.py: f.write(data)
```

Come possiamo vedere, per i file di testo viene mostrato il contenuto della linea che verifica il pattern cercato, mentre per i file binari viene solamente indicato che il pattern è verificato.

head

Il comando `head` visualizza la parte iniziale di un file di testo. Per default vengono mostrate le prime 10 linee:

```
$ head /usr/local/lib/python3.3/LICENSE.txt  
A. HISTORY OF THE SOFTWARE
```

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <http://www.cwi.nl>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <http://www.cnri.reston.va.us>)

L'opzione `-n` consente di specificare il numero di linee da visualizzare:

```
$ head -n 5 /usr/local/lib/python3.3/LICENSE.txt  
A. HISTORY OF THE SOFTWARE
```

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <http://www.cwi.nl>) in the Netherlands

ln

Il comando `ln` crea collegamenti simbolici e collegamenti fisici a file e directory. Consideriamo, ad esempio, la seguente struttura di directory:

```
$ tree top/  
top/  
├── nested1  
│   └── file1  
└── nested2
```

2 directories, 1 file

In *top/nested1* è presente il seguente file *file1*:

```
$ more top/nested1/file1  
Contenuto di nested1/file1
```

Se vogliamo creare un file *nested2/file2* che sia un collegamento simbolico al file *nested1/file1*, usiamo il comando `ln` con lo switch `-s`:

```
$ ln -s $PWD/top/nested1/file1 top/nested2/file2  
$ ls -l top/nested2/file2  
lrwxrwxrwx 1 marco marco 34 May 23 21:56 top/nested2/file2 -> /home/marco/temp/top/nested1/file1  
$ more top/nested2/file2  
Contenuto di nested1/file1
```

ls

Il comando `ls` (abbreviazione dell'inglese *list segments*, “elenca segmenti”) elenca informazioni su file e il contenuto delle directory. Se non passiamo alcun argomento, mostra il contenuto della directory corrente:

```
$ ls
abc.py __init__.py __main__.py __pycache__
```

Se vogliamo sapere quali sono i file e quali le directory, possiamo passare a `ls` l'opzione `-F`. In questo modo, il nome delle directory terminerà con un carattere di backslash:

```
$ ls -F
abc.py __init__.py __main__.py __pycache__/
```

In questo caso, quindi, `__pycache__` è una directory. Per visualizzare un file per linea si utilizza lo switch `-l` (uno):

```
$ ls -l -F
abc.py
__init__.py
__main__.py
__pycache__/
```

Se a `ls` passiamo come argomento il percorso di una directory, allora mostra il contenuto di questa:

```
$ ls /usr/local/lib/python3.3/collections # Percorso assoluto
abc.py __init__.py __main__.py __pycache__
$ ls __pycache__/ # Percorso relativo
abc.cpython-33.pyc abc.cpython-33.pyo __init__.cpython-33.pyc __init__.cpython-33.pyo __main__.cpython-33.pyc
__main__.cpython-33.pyo
```

Spesso il comando `ls` viene utilizzato in combinazione con il metacarattere `*`, che rappresenta uno o più caratteri. Ad esempio, nel seguente caso vengono mostrati i file con suffisso `.pyc` contenuti nella directory `__pycache__`:

```
$ ls __pycache__/*.pyc
__pycache__/abc.cpython-33.pyc __pycache__/__init__.cpython-33.pyc __pycache__/__main__.cpython-33.pyc
```

Nell'esempio seguente vengono mostrati i file della directory corrente che contengono il carattere `i` nel loro nome:

```
$ ls *i*
__init__.py __main__.py
```

L'opzione `-l` (elle minuscola) mostra varie informazioni sui file, come i permessi, il proprietario, il gruppo e la data di ultima modifica:

```
$ ls -l
total 72
-rw-r--r-- 1 root root 19147 Apr 12 16:27 abc.py
-rw-r--r-- 1 root root 41704 Apr 12 16:27 __init__.py
-rw-r--r-- 1 root root 1275 Apr 12 16:27 __main__.py
drwxr-xr-x 2 root root 4096 Feb 21 18:25 __pycache__
```

La dimensione dei file è espressa in byte. Per visualizzarla in un formato *human readable* si utilizza lo switch `-h`:

```
$ ls -lh
total 72K
-rw-r--r-- 1 root root 19K Apr 12 16:27 abc.py
-rw-r--r-- 1 root root 41K Apr 12 16:27 __init__.py
-rw-r--r-- 1 root root 1.3K Apr 12 16:27 __main__.py
drwxr-xr-x 2 root root 4.0K Feb 21 18:25 __pycache__
```

Per visualizzare la data e l'ora di ultima modifica con uno stile diverso si utilizza l'opzione `--time-style`. Possibili argomenti sono `iso`, `long-iso`, `full-iso`:

```
$ ls -l --time-style=iso
total 72
-rw-r--r-- 1 root root 19147 04-12 16:27 abc.py
-rw-r--r-- 1 root root 41704 04-12 16:27 __init__.py
-rw-r--r-- 1 root root 1275 04-12 16:27 __main__.py
drwxr-xr-x 2 root root 4096 02-21 18:25 __pycache__
$ ls -l --time-style=long-iso
total 72
-rw-r--r-- 1 root root 19147 2013-04-12 16:27 abc.py
-rw-r--r-- 1 root root 41704 2013-04-12 16:27 __init__.py
-rw-r--r-- 1 root root 1275 2013-04-12 16:27 __main__.py
drwxr-xr-x 2 root root 4096 2013-02-21 18:25 __pycache__
$ ls -l --time-style=full-iso
total 72
-rw-r--r-- 1 root root 19147 2013-04-12 16:27:11.977645657 +0200 abc.py
-rw-r--r-- 1 root root 41704 2013-04-12 16:27:11.977645657 +0200 __init__.py
-rw-r--r-- 1 root root 1275 2013-04-12 16:27:11.977645657 +0200 __main__.py
drwxr-xr-x 2 root root 4096 2013-02-21 18:25:53.929542436 +0100 __pycache__
```

Se vogliamo visualizzare solamente alcune colonne mostrate da `ls`, possiamo

mettere l'output di `ls` in *pipe* (simbolo `|`) con il comando `cut`:

```
$ ls -l --time-style=full-iso abc.py | cut -d ' ' -f7  
16:27:11.977645657
```

L'opzione `-d` viene utilizzata per indicare il delimitatore delle colonne, che in questo caso è uno spazio, mentre l'opzione `-f` serve per specificare la colonna che vogliamo selezionare:

```
$ ls -l --time-style=full-iso | cut -d ' ' -f2  
72  
1  
1  
1  
2
```

Ecco qualche altro esempio:

```
$ ls -l --time-style=full-iso abc.py | cut -d ' ' -f6,7 # Data e ora  
2013-04-12 16:27:11.977645657  
$ ls -l --time-style=full-iso | tr -s ' ' | cut -d ' ' -f 6,7  
  
2013-04-12 16:27:11.977645657  
2013-04-12 16:27:11.977645657  
2013-04-12 16:27:11.977645657  
2013-02-21 18:25:53.929542436
```

Il significato del comando `tr` con l'opzione `-s` è spiegato nella sezione dedicata al comando `cut`. Se utilizziamo lo switch `-R` (`--recursive`), allora viene elencato il contenuto dell'intero albero di directory. Consideriamo, ad esempio, la seguente struttura di directory:

```
$ tree top  
top  
├── myfile  
├── nested1  
│   └── file1  
└── nested2  
    └── file2
```

2 directories, 3 files

Ecco come appare l'output di `ls -R`:

```
$ ls -R top  
top:
```

```
myfile nested1 nested2
```

```
top/nested1:  
file1
```

```
top/nested2:  
file2
```

Si può ottenere un risultato analogo utilizzando il metacarattere *:

```
$ ls top/*  
top/myfile
```

```
top/nested1:  
file1
```

```
top/nested2:  
file2
```

Se non si vuole visualizzare il contenuto delle sotto-directory, si utilizza lo switch -d:

```
$ ls top/* -d  
top/myfile top/nested1 top/nested2
```

mkdir

Il comando `mkdir` crea una o più directory:

```
$ mkdir foo # Crea la directory foo
$ mkdir foo/dirA foo/dirB # Crea le sotto-directory dirA e dirB di foo
$ tree foo
foo
├── dirA
└── dirB
```

2 directories, 0 files

more

Il comando `more` mostra il contenuto di uno o più file di testo (o dello standard input) su un terminale testuale, visualizzandolo una pagina per volta, permettendo di scorrelo in avanti e all'indietro (solo in caso di file) e di effettuare ricerche tramite espressioni regolari.

Ad esempio, nel caso seguente usiamo `more` per visualizzare il contenuto del file `.vimrc`:

```
$ more ~/.vimrc # Equivalente a `$ more /home/marco/.vimrc`
syntax enable
set number
filetype plugin indent on
set ai ts=4 sts=4 et sw=4
set backspace=2
set nobackup
imap { }<esc>i
set makeprg=g++\ %<.cpp\ -o\ %<
map <f5> :w<CR>:!python %<CR>
map <f6> :w<CR>:!make %<CR>
map <C-n> :tabn<CR>
map <C-p> :tabp<CR>
```

Il carattere di *tilde* (`~`) rappresenta il percorso della home directory:

```
$ ~
bash: /home/marco: Is a directory
```

Quindi scrivere `more ~/.vimrc` è equivalente a `more /home/marco/.vimrc`.

mv

Il comando `mv` (abbreviazione dell'inglese *move*, “sposta”) sposta file e directory:

```
$ tree top
```

```
top
├── nested1
│   └── file1
└── nested2
    └── file2
```

2 directories, 2 files

```
$ mv top/nested2/file2 top/nested1/
```

```
$ tree top
```

```
top
├── nested1
│   ├── file1
│   └── file2
└── nested2
```

2 directories, 2 files

Non esiste un comando `rename` perché si ottiene il medesimo comportamento quando si effettua uno spostamento all'interno della stessa directory:

```
$ mv top/nested1/file1 top/nested1/foo
```

```
$ tree top
```

```
top
├── nested1
│   ├── file2
│   └── foo
└── nested2
```

2 directories, 2 files

pwd

Il comando `pwd` (abbreviazione dell'inglese *print working directory*, “stampa la directory corrente”) mostra sullo standard output il percorso assoluto della directory corrente:

```
$ cd /home/marco/temp/  
$ pwd  
/home/marco/temp
```

rm

Il comando `rm` (abbreviazione dall'inglese *remove*, “rimuovi”) cancella file e directory dal file system. Ad esempio, consideriamo la seguente directory:

```
$ ls -F
foodir/ foofile
```

Per rimuovere il file *foofile* diamo il comando `rm foofile`:

```
$ rm foofile
$ ls -F
foodir/
```

Il comando `rm` per default non rimuove le directory:

```
$ rm foodir/
rm: cannot remove `foodir/': Is a directory
```

Per rimuovere una directory dobbiamo usare lo switch `-r` o `-R`:

```
$ rm foodir/ -r
$ ls
$
```

sed

Il comando `sed` (abbreviazione dell'inglese *stream editor*, “editor di flusso”) consente il filtraggio e la manipolazione di testi. Nel libro lo abbiamo utilizzato con l'opzione `-i` (`--in-place`), in modo da cercare del testo all'interno di un file con lo scopo di sostituirlo con altro testo. Ad esempio, consideriamo il seguente file:

```
$ more myfile
prima linea
seconda linea
terza linea
```

Se vogliamo sostituire la parola `linea` con `Linea`, chiamiamo `sed` nel seguente modo:

```
$ sed -i 's/linea/Linea/' myfile
```

Ecco, infatti, il nuovo contenuto del file:

```
$ more myfile
prima Linea
seconda Linea
terza Linea
```

La `s` all'inizio della stringa `'s/linea/Linea/'` significa *substitution* e indica, per l'appunto, che `linea` deve essere sostituita con `Linea`.

source

Il comando `source` legge i comandi dal file che gli viene passato come argomento e li esegue nella shell corrente. Consideriamo, ad esempio, il seguente script bash, che stampa un messaggio:

```
$ more myscript.bash  
echo "Python 3.3 :)"
```

Possiamo eseguirlo con `source`:

```
$ source myscript.bash  
Python 3.3 :)
```

tail

Il comando `tail` (il termine *tail* in inglese significa “coda, estremità”) mostra sullo standard output le ultime linee di dati provenienti da uno o più file di testo o dallo standard input. Per default vengono mostrate le ultime 10 linee:

```
$ tail INSTALL
```

AS AN ALTERNATIVE, you can just copy the entire "django" directory to Python's site-packages directory, which is located wherever your Python installation lives. Some places you might check are:

```
/usr/lib/python2.7/site-packages (Unix, Python 2.7)
```

```
/usr/lib/python2.6/site-packages (Unix, Python 2.6)
```

```
C:\PYTHON\site-packages (Windows)
```

For more detailed instructions, see `docs/intro/install.txt`.

Se si vuole indicare il numero di linee da visualizzare, si utilizza lo switch `-n`:

```
$ tail INSTALL -n 1
```

For more detailed instructions, see `docs/intro/install.txt`.

tar

Il comando `tar` (acronimo per *tape archive*) permette di generare dei file utili per l'archiviazione e il backup, utilizzando il formato omonimo (*.tar*). Negli archivi *.tar* sono presenti tutte le informazioni per ricostruire correttamente la gerarchia di directory originale con tutto il suo contenuto, comprese le informazioni del file system, come utente, gruppo e permessi, data e ora. Supponiamo di avere la seguente struttura di directory:

```
$ tree top
top
├── myfile
├── nested1
│   └── file1
├── nested2
└── file2
```

2 directories, 3 files

Per creare sul file-system un archivio *.tar* di essa, dobbiamo usare le opzioni `c` (create) e `f` (file), seguite dal nome che vogliamo dare all'archivio e dal nome della directory da archiviare:

```
$ tar cf top.tar top
$
```

Se vogliamo avere un output dal comando `tar`, usiamo lo switch `v` (*verbose*):

```
$ tar cfv top.tar top
top/
top/myfile
top/nested1/
top/nested1/file1
top/nested2/
top/nested2/file2
```

Gli archivi `tar` per default non sono compressi. Se vogliamo creare un archivio compresso, dobbiamo specificare delle opzioni di compressione, le quali richiamano dei programmi esterni che si occupano, appunto, di comprimere l'archivio. Ad esempio, per richiamare `gzip` si utilizza lo switch `z`:

```
$ tar cfz top.tar.gz top
```

Come possiamo vedere, l'archivio è stato effettivamente compresso:

```
$ ls -l top.tar.gz
-rw-r--r-- 1 marco marco 244 May 25 17:05 top.tar.gz
$ ls -l top.tar
-rw-r--r-- 1 marco marco 10240 May 25 16:58 top.tar
```

Per comprimere usando `bzip2` si usa lo switch `j`:

```
$ tar cfj top.tar.bz2 top
marco@buttu-oac ~/temp $ ls -l top.tar.bz2
-rw-r--r-- 1 marco marco 254 May 25 17:09 top.tar.bz2
```

Per estrarre un archivio si usa lo switch `x` (`--extract`) al posto dello switch `c`:

```
$ tar xvf top.tar
top/
top/myfile
top/nested1/
top/nested1/file1
top/nested2/
top/nested2/file2
```

Se l'archivio è compresso, dobbiamo anche decomprimerlo richiamando il programma corretto, quindi usiamo lo switch `z` se è stato compresso con `gzip` oppure lo switch `j` se è stato compresso con `bzip2`:

```
$ tar zxvf top.tar.gz
top/
top/myfile
top/nested1/
top/nested1/file1
top/nested2/
top/nested2/file2
$ tar jxvf top.tar.bz2
top/
top/myfile
top/nested1/
top/nested1/file1
top/nested2/
top/nested2/file2
```

Lo switch `t` consente di elencare il contenuto di un archivio, senza estrarlo:

```
$ tar tf top.tar
top/
top/myfile
```

```
top/nested1/  
top/nested1/file1  
top/nested2/  
top/nested2/file2  
$ tar ztf top.tar.gz  
top/  
top/myfile  
top/nested1/  
top/nested1/file1  
top/nested2/  
top/nested2/file2  
$ tar jtf top.tar.bz2  
top/  
top/myfile  
top/nested1/  
top/nested1/file1  
top/nested2/  
top/nested2/file2
```

Se vogliamo vedere il contenuto di un file presente in un archivio possiamo usare lo switch `-o`, che consente di stampare sullo standard output il contenuto del file. Ad esempio, se vogliamo visualizzare il contenuto del file *top/nested1/file1*:

```
$ tar xfO top.tar top/nested1/file1  
Contenuto di nested1/file1  
$ tar zxfO top.tar.gz top/nested1/file1  
Contenuto di nested1/file1  
$ tar jxfO top.tar.bz2 top/nested1/file1  
Contenuto di nested1/file1
```


time

Il comando `time` avvia un programma e, quando esso termina, visualizza sullo standard error il tempo impiegato per eseguirlo:

```
$ time ls > /dev/null
```

```
real 0m0.002s
user 0m0.000s
sys 0m0.000s
```

In questo esempio, abbiamo reindirizzato l'output di `ls` su `/dev/null`, per cui possiamo vedere a video il solo risultato di `time`, dato che questo viene inviato allo standard error. Il tempo è suddiviso in tre parti:

1. tempo di esecuzione totale: tempo trascorso dall'avvio al termine del programma;
2. tempo di CPU utente: tempo impiegato dalla CPU per eseguire le istruzioni non di sistema del programma (user mode);
3. tempo di CPU di sistema: tempo impiegato dalla CPU per eseguire le istruzioni di sistema del programma (kernel mode).

Ovviamente la somma del tempo di CPU di sistema e del tempo di CPU utente è sempre inferiore o uguale al tempo totale di esecuzione, in quanto, per ottenere quest'ultimo, va sommato ai primi due il tempo in cui il programma rimane inattivo, ad esempio perché la CPU è impegnata a eseguire altri programmi.

touch

Il comando `touch` permette di impostare la data e l'ora di ultima modifica e/o di ultimo accesso di uno o più file e directory. Consideriamo, ad esempio, il file seguente:

```
$ ls -l foo.py
-rw-r--r-- 1 marco marco 42 Apr 30 13:50 foo.py
```

Ecco cosa accade se diamo il comando `touch foo.py`:

```
$ touch foo.py
$ ls -l foo.py
-rw-r--r-- 1 marco marco 42 May 20 21:55 foo.py
```

Se il file non esiste, ne viene creato uno nuovo:

```
$ ls -l moo.py # Il file `moo.py` non esiste
ls: cannot access moo.py: No such file or directory
$ touch moo.py # Viene creato il file `moo.py`
$ ls -l moo.py
-rw-r--r-- 1 marco marco 0 May 20 21:57 moo.py
```

tree

Il comando `tree` consente di visualizzare directory e sottodirectory (e opzionalmente i file) utilizzando una struttura ad albero, in modo da avere una visione della disposizione/posizione di file e directory:

```
$ tree /usr/local/lib/python3.3/collections/  
/usr/local/lib/python3.3/collections/
```

```
├── abc.py  
├── __init__.py  
├── __main__.py  
├── __pycache__  
├── abc.cpython-33.pyc  
├── abc.cpython-33.pyo  
├── __init__.cpython-33.pyc  
├── __init__.cpython-33.pyo  
├── __main__.cpython-33.pyc  
└── __main__.cpython-33.pyo
```

1 directory, 9 files

Il comando `tree` è presente in tutte le versioni di Microsoft Windows e MS-DOS. Esiste una sua versione anche per Linux, che tipicamente, però, non fa parte delle applicazioni installate di default nel sistema. Il sito ufficiale è: <http://mama.indstate.edu/users/ice/tree/>.

WC

Il comando `wc` (abbreviazione dell'inglese *word count*, “conteggio delle parole”) produce sullo standard output un conteggio delle linee, delle parole e dei byte che costituiscono uno o più file di testo specificati come parametri (o dei dati provenienti dallo standard input).

Ad esempio, consideriamo il seguente file:

```
$ more myfile
questa è la prima linea
questa è la seconda linea
questa è la terza linea
```

Per default `wc` mostra, rispettivamente, il numero di newline, il numero di parole e il numero di byte presenti nel file:

```
$ wc myfile
3 15 77 myfile
```

Usando gli switch `-c` (*byte counts*), `-m` (*character counts*), `-l` (*newline counts*) e `-w` (*words*), è possibile visualizzare, rispettivamente, il numero di byte, quello di caratteri, quello di newline e quello di parole:

```
$ wc -c myfile
77 myfile
$ wc -m myfile
74 myfile
$ wc -l myfile
3 myfile
$ wc -w myfile
15 myfile
```

wget

Il comando `wget` consente di gestire i download da linea di comando. Ad esempio, per scaricare la pagina della PEP-0008:

```
$ wget http://www.python.org/dev/peps/pep-0008/
--2013-05-25 12:10:31-- http://www.python.org/dev/peps/pep-0008/
Resolving www.python.org (www.python.org)... 82.94.164.162, 2001:888:2000:d:a2
Connecting to www.python.org (www.python.org)|82.94.164.162|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 60722 (59K) [text/html]
Saving to: `index.html.1'

100%[=====>] 60,722 153K/s in 0.4s

2013-05-25 12:10:32 (153 KB/s) - `index.html.1' saved [60722/60722]
```

Se vogliamo che `wget` operi in modo “silenzioso”, usiamo lo switch `-q` (`--quiet`):

```
$ wget -q http://www.python.org/dev/peps/pep-0008/ # Non viene mostrato l'output
$
```

which

Il comando `which` 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
```

zip

Il comando `zip` crea un archivio compresso in formato ZIP. Consideriamo, ad esempio, la seguente directory:

```
$ tree mydir/
mydir/
├── dirA
│   ├── afoofile.txt
│   └── foo
├── dirB
│   └── bfoofile.txt
├── myfile.txt
└── myfoo.txt
```

2 directories, 5 files

Ecco come creare un archivio ZIP contenente i due file di testo presenti in *mydir*:

```
$ zip myzip.zip mydir/*.txt
adding: mydir/myfile.txt (deflated 10%)
adding: mydir/myfoo.txt (stored 0%)
```

Proviamo ora a creare un archivio di una directory:

```
$ zip myzip.zip mydir/
adding: mydir/ (stored 0%)
```

Come possiamo vedere dall'output, il contenuto della directory non è stato archiviato. Se vogliamo creare un archivio contenente una intera struttura di directory, dobbiamo usare lo switch `-r` (`--recurse-paths`):

```
$ zip myzip.zip mydir/ -r
updating: mydir/myfile.txt (deflated 10%)
updating: mydir/myfoo.txt (stored 0%)
updating: mydir/ (stored 0%)
  adding: mydir/dirA/ (stored 0%)
    adding: mydir/dirA/afoofile.txt (deflated 19%)
    adding: mydir/dirA/foo (stored 0%)
  adding: mydir/dirB/ (stored 0%)
    adding: mydir/dirB/bfoofile.txt (deflated 19%)
```

I metacaratteri

La shell Unix riconosce alcuni caratteri speciali, chiamati metacaratteri, che possono comparire nelle istruzioni eseguite da linea di comandi. Questi caratteri sono riportati in Tabella A.1.

Tabella A.1 - Tipi di conversione disponibili per formattare le stringhe di testo.

Metacarattere	Significato
>	Redirezione dell'output
>>	Redirezione dell'output (append)
<	Redirezione dell'input
2>	Redirezione dei messaggi di errore
*	Rappresenta zero o più caratteri
?	Rappresenta un solo carattere
[]	Rappresenta ogni carattere racchiuso tra le parentesi
`cmd`	Sostituisce il comando
\$(cmd)	Sostituisce il comando
	Pipe
;	Delimitatore di comandi
	OR logico
&&	AND logico
()	Raggruppa comandi e sequenze di comandi
&	Esegue i comandi in background
#	Inizio di un commento
\$	Espansione del valore di una variabile
\	Evita che il successivo carattere non venga interpretato come sequenza di escape
<<	Redirezione dell'input

Quando l'utente invia un'istruzione, la shell effettua la ricerca di eventuali metacaratteri, che poi processa assegnando a essi un significato speciale. Una volta processati tutti i metacaratteri, viene eseguita l'istruzione.

In questa sezione vedremo solamente il significato dei metacaratteri utilizzati nel libro.

Le wildcard

Alcuni metacaratteri vengono utilizzati per rappresentare altri caratteri all'interno dei nomi dei file e dei percorsi. Questi metacaratteri, detti *wildcard*, sono: *, ? e [].

Il metacarattere *, come probabilmente sappiamo, rappresenta uno o più caratteri:

```
$ ls /usr/local/lib/python3.3/collections/  
abc.py __init__.py __main__.py __pycache__  
$ ls -l /usr/local/lib/python3.3/collections/*.py # Mostra solo i file i cui nomi terminano con .py  
/usr/local/lib/python3.3/collections/abc.py  
/usr/local/lib/python3.3/collections/__init__.py  
/usr/local/lib/python3.3/collections/__main__.py  
$ ls -l /usr/local/lib/python3.3/collections/*i*.py # Mostra solo i file che contengono una `i` nel nome  
/usr/local/lib/python3.3/collections/__init__.py  
/usr/local/lib/python3.3/collections/__main__.py
```

Il metacarattere ? rappresenta un solo carattere:

```
$ ls /dev/sda*  
/dev/sda /dev/sda1 /dev/sda2 /dev/sda3 /dev/sda4 /dev/sda5 /dev/sda6 /dev/sda7 /dev/sda8  
$ ls /dev/sda?  
/dev/sda1 /dev/sda2 /dev/sda3 /dev/sda4 /dev/sda5 /dev/sda6 /dev/sda7 /dev/sda8
```

Il metacarattere [] consente di specificare un gruppo di caratteri da rappresentare (o da non rappresentare):

```
$ ls /dev/[a-r]da?  
ls: cannot access /dev/[a-r]da?: No such file or directory  
$ ls /dev/[a-s]da?  
/dev/sda1 /dev/sda2 /dev/sda3 /dev/sda4 /dev/sda5 /dev/sda6 /dev/sda7 /dev/sda8
```

Redirezione

Talvolta può risultare molto utile indirizzare l'output generato da un comando su un file. Questo può essere fatto facendo seguire il comando dal metacarattere >, detto *operatore di redirezione dell'output*. Ad esempio, nel caso seguente scriviamo `print(__name__)` sul file *myfile.py*:

```
$ echo "print(__name__)" > myfile.py
```

Ecco il contenuto del file da indirizzare:

```
$ more myfile.py  
print(__name__)
```

Nel seguente esempio scriviamo sul file *mylist* l'elenco dei file con estensione *.py* presenti nella directory */usr/bin*:

```
$ ls /usr/bin/*.py > mylist
```

Ecco il contenuto del file:

```
$ more mylist
/usr/bin/miniterm.py
/usr/bin/pilconvert.py
/usr/bin/pildriver.py
/usr/bin/pilfile.py
/usr/bin/pilfont.py
/usr/bin/pilprint.py
```

Se il file esiste viene sovrascritto:

```
$ echo "Fooooo" > mylist
$ more mylist
Fooooo
```

Se si vuole conservare il contenuto e scrivere in coda al file, si usa l'operatore *>>*. Questo si dice che fa un *append* sul file:

```
$ echo "Python 3" >> mylist
$ more mylist
Fooooo
Python 3
```

La pipe

Il metacarattere *|*, chiamato *pipe*, consente di concatenare più programmi fra loro e fare in modo che lo standard output di uno diventi lo standard input di un altro.

Consideriamo, ad esempio, il seguente output del comando *ls*:

```
$ ls -l ~/.vimrc
-rwxr-x--- 1 marco marco 1733 May 25 08:35 /home/marco/.vimrc
```

Se vogliamo visualizzare solamente la dimensione del file, possiamo usare la pipe mandando l'output di *ls* in ingresso a *cut*:

```
$ ls -l ~/.vimrc | cut -d ' ' -f 5
1733
```

Vediamo un altro esempio significativo. Supponiamo di voler visualizzare il numero di file presenti all'interno di una directory. Una soluzione è quella di mandare l'output di `ls` in ingresso al comando `wc`:

```
$ ls /usr/local/lib/python3.3/*.py | wc -w  
156
```

Variabili d'ambiente

Nel libro abbiamo usato alcune variabili d'ambiente, come `PWD` e `HOME`.

PWD

La variabile d'ambiente `PWD` ha come valore il percorso completo della directory corrente:

```
$ echo $PWD
/home/marco/myenv
$ pwd
/home/marco/myenv
$ ls $PWD
bin include lib pyenv.cfg share
$ ls $PWD/lib
python3.3
```

HOME

La variabile d'ambiente `HOME` ha come valore il percorso della home directory dell'utente corrente. Ad esempio, se stiamo lavorando come utente `marco`:

```
$ echo $USER # Mostra l'utente
marco
```

la variabile d'ambiente `HOME` è impostata al percorso della home di `marco`:

```
$ echo $HOME
/home/marco
```

Se cambiamo utente, cambia quindi anche il valore di `HOME`:

```
$ su offline
$ echo $USER
offline
$ echo $HOME
/home/offline
```

Appendice B

Principali punti di rottura tra Python 2 e Python 3

In questo capitolo elencheremo le principali **incompatibilità** tra Python 2 e 3 e daremo alcune indicazioni su come fare il **porting** del codice tra le due versioni. Al termine della sezione sono riportati alcuni utili riferimenti.

Incompatibilità tra le due versioni

Nella sezione *Sviluppo di Python* del Capitolo 1 abbiamo detto che per le *major versions* non è garantita la *retrocompatibilità*, e infatti in Python 3 vi sono diverse *incompatibilità* con Python 2, per cui una parte del codice scritto per Python 2 non può essere eseguito con Python 3, oppure può essere eseguito ma dà luogo a un risultato diverso. In questa sezione elencheremo le principali incompatibilità tra le due versioni.

Print è una funzione built in Python 3 e una istruzione in Python 2

In Python 2 `print` è una istruzione e quindi anche una parola chiave:

```
>>> help('keywords') # Python 2
```

Here is a list of the Python keywords. Enter any keyword to get more help.

<code>and</code>	<code>elif</code>	<code>if</code>	<code>print</code>
<code>as</code>	<code>else</code>	<code>import</code>	<code>raise</code>
<code>assert</code>	<code>except</code>	<code>in</code>	<code>return</code>
<code>break</code>	<code>exec</code>	<code>is</code>	<code>try</code>
<code>class</code>	<code>finally</code>	<code>lambda</code>	<code>while</code>
<code>continue</code>	<code>for</code>	<code>not</code>	<code>with</code>
<code>def</code>	<code>from</code>	<code>or</code>	<code>yield</code>
<code>del</code>	<code>global</code>	<code>pass</code>	

In Python 3, invece, `print` è una etichetta che fa riferimento a una funzione built-in:

```
>>> print # Python 3
<built-in function print>
```

Nella PEP-3105 sono discusse le motivazioni che hanno portato a questa modifica del linguaggio. Questo cambiamento dà luogo a una incompatibilità, poiché il seguente codice eseguito in Python 2:

```
>>> print 33 # Python 2
33
```

dà luogo a un errore di sintassi quando eseguito in Python 3:

```
>>> print 33
```

```
File "<stdin>", line 1
print 33
^
SyntaxError: invalid syntax
>>> print(33) # Stampa del numero 33 in Python 3
33
```

Divisione vera e divisione floor

In Python 2 l'operatore `/` effettua una divisione *floor* tra due numeri, ovvero arrotonda il risultato all'intero inferiore:

```
>>> -5 / 2 # Python 2
-3
```

In Python 3, invece, effettua una divisione vera:

```
>>> -5 / 2
-2.5
```

Nella PEP-0238 sono riportate le motivazioni che hanno condotto a questo cambiamento.

Le funzioni built-in `input()` e `raw_input()`

In Python 2 vi è la funzione built-in `input()` che *valuta* ciò che le viene passato e lo restituisce:

```
>>> a = input('Digita qualcosa: ')
Digita qualcosa: 2.5
>>> type(a)
<type 'float'>
>>> a = input('Digita qualcosa: ')
Digita qualcosa: "ciao"
>>> a
'ciao'
>>> a = input('Digita qualcosa: ')
Digita qualcosa: ciao
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'ciao' is not defined
```

In Python 3 la funzione built-in `input()` restituisce sempre una stringa e si comporta allo stesso modo della funzione `raw_input()` di Python 2, che non esiste più:

```
>>> a = input('Digita qualcosa: ')
```

```

Digita qualcosa: 2.5
>>> type(a)
<class 'str'>
>>> a = input('Digita qualcosa: ')
Digita qualcosa: ciao
>>> a
'ciao'
>>> a = raw_input('Digita qualcosa: ')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'raw_input' is not defined

```

Per maggiori dettagli si veda la PEP- 3111.

La funzione built-in `file()` non esiste più in Python 3

In Python 2 un file ad alto livello può essere aperto sia con la funzione built-in `open()` sia con la funzione built-in `file()`:

```

>>> f = file('myfile', 'w') # Python 2
>>> type(f)
<type 'file'>
>>> f = open('myfile', 'w')
>>> type(f)
<type 'file'>

```

In Python 3, invece, la funzione built-in `file()` non esiste più e un file può essere aperto ad alto livello solo con `open()`:

```

>>> f = file('myfile', 'w') # Python 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'file' is not defined
>>> f = open('myfile', 'w')
>>> type(f)
<class '_io.TextIOWrapper'>

```

Le funzioni built-in `range()`, `filter()`, `map()` e `zip()`

Le funzioni built-in `range()`, `filter()`, `map()` e `zip()` restituiscono degli iteratori in Python 3 e dei tipi built-in in Python 2.

La funzione built-in `range()` restituisce una lista in Python 2 e un oggetto di tipo `range` in Python 3. Ciò significa che il codice scritto in Python 2 considerando il risultato di `range()` come lista, come ad esempio il seguente:

```

>>> range(4) + [4, 5, 6] # Python 2
[0, 1, 2, 3, 4, 5, 6]

```


dà luogo a un errore in Python 3:

```
>>> range(4) + [4, 5, 6] # Python 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'range' and 'list'
>>> list(range(4)) + [4, 5, 6]
[0, 1, 2, 3, 4, 5, 6]
```

La funzione built-in `filter()` in Python 2 restituisce la sequenza passata come secondo argomento:

```
>>> def foo(x):
...     return True if x % 2 == 0 else False
...
>>> ['a', 'b'] + filter(foo, [1, 2, 3, 4, 5, 6])
['a', 'b', 2, 4, 6]
>>> filter(foo, (1, 2, 3, 4, 5, 6))
(2, 4, 6)
```

mentre in Python 3 restituisce un iteratore di tipo `filter`:

```
>>> ['a', 'b'] + filter(foo, [1, 2, 3, 4, 5, 6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "filter") to list
>>> ['a', 'b'] + list(filter(foo, [1, 2, 3, 4, 5, 6]))
['a', 'b', 2, 4, 6]
```

La funzione built-in `map()` in Python 2 restituisce una lista:

```
>>> map(foo, (1, 2, 3, 4))
[1, 4, 9, 16]
```

mentre in Python 3 restituisce un iteratore:

```
>>> map(foo, (1, 2, 3, 4))
<map object at 0xb737aecc>
>>> list(map(foo, (1, 2, 3, 4)))
[1, 4, 9, 16]
```

La funzione built-in `zip()` restituisce una lista in Python 2:

```
>>> zip((1, 2, 3), (4, 4, 4))
[(1, 4), (2, 4), (3, 4)]
```

e un iteratore di tipo `zip` in Python 3:

```
>>> zip((1, 2, 3), (4, 4, 4))
<zip object at 0xb71e8fac>
>>> list(zip((1, 2, 3), (4, 4, 4)))
[(1, 4), (2, 4), (3, 4)]
```

I metodi `keys()`, `values()` e `items()` dei dizionari

I metodi `dict.keys()`, `dict.values()` e `dict.items()` di un dizionario restituiscono una lista in Python 2:

```
>>> d = {1: 'uno', 2: 'due', 3: 'tre'}
>>> d.keys()
[1, 2, 3]
>>> d.values()
['uno', 'due', 'tre']
>>> d.items()
[(1, 'uno'), (2, 'due'), (3, 'tre')]
```

mentre in Python 3 restituiscono degli oggetti di tipo `dict_keys`, `dict_values` e `dict_items`:

```
>>> d = {1: 'uno', 2: 'due', 3: 'tre'}
>>> d.keys()
dict_keys([1, 2, 3])
>>> d.values()
dict_values(['uno', 'due', 'tre'])
>>> d.items()
dict_items([(1, 'uno'), (2, 'due'), (3, 'tre')])
```

List comprehension

In Python 2 era lecito utilizzare la seguente sintassi in una list comprehension:

```
>>> [i for i in 1, 2, 3] # Python 2
[1, 2, 3]
```

Con gli interi 1, 2 e 3 veniva costruita una tupla sulla quale la list comprehension andava a iterare:

```
>>> def foo():
... [i for i in 1, 2, 3]
...
>>> dis.dis(foo)
2 0 BUILD_LIST 0
3 LOAD_CONST 4 ((1, 2, 3))
6 GET_ITER
>> 7 FOR_ITER 12 (to 22)
10 STORE_FAST 0 (i)
```

```
13 LOAD_FAST 0 (i)
16 LIST_APPEND 2
19 JUMP_ABSOLUTE 7
>> 22 POP_TOP
23 LOAD_CONST 0 (None)
26 RETURN_VALUE
```

In Python 3 questo non è più possibile:

```
>>> [i for i in 1, 2, 3]
File "<stdin>", line 1
[i for i in 1, 2, 3]
^
SyntaxError: invalid syntax
>>> [i for i in (1, 2, 3)]
[1, 2, 3]
```

Stringhe di testo e byte

In Python 2 non c'è distinzione tra stringhe di testo e di byte:

```
>>> b = b"ciao" # Python 2
>>> type(b)
<type 'str'>
>>> b"ciao" + "miao"
'ciaomiao'
```

In Python 3, invece, una stringa di `b"ciao"` è una stringa di byte:

```
>>> b = b'ciao' # Python 3
>>> type(b)
<class 'bytes'>
>>> b'ciao' + "miao"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
```

Unificazione tra interi e long

In Python 2 esiste il tipo numerico `long`:

```
>>> type(44L)
<type 'long'>
```

In Python 3 c'è solo il tipo `int` a rappresentare gli interi:

```
>>> 44L
File "<stdin>", line 1
```

44L

^

SyntaxError: invalid syntax

In Python 3 non c'è più la funzione built-in apply()

La funzione built-in `apply()`, che in Python 2 chiamava l'oggetto passatole come primo argomento:

```
>>> def foo(a, b):
...     print a, b
...
>>> apply(foo,(10, 20)) # Python 2
10 20
```

non esiste più in Python 3:

```
>>> apply # Python 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'apply' is not defined
```

In Python 2 `exec` è una parola chiave, mentre in Python 3 è una funzione built-in

In Python 2 `exec` è una istruzione:

```
>>> a, b = 1, 2
>>> exec "print a, b"
1 2
```

mentre in Python 3 è una funzione built-in:

```
>>> a, b = 1, 2
>>> exec "print(a, b)"
File "<stdin>", line 1
exec "print(a, b)"
^
SyntaxError: invalid syntax
>>> exec("print(a, b)")
1 2
```

Uso inconsistente di spazi e tabulazioni

In Python 2, anche se sconsigliato, è possibile passare da spazi a tab e viceversa, cosa che invece non è consentita in Python 3:

```
$ more foo.py
# File: foo.py
```

```
for i in range(2):
if i % 2 == 0: # Indentazione con spazi
print(i) # Indentazione con tab
print(i) # Indentazione con tab
else: # Indentazione con spazi
pass
$ python2.7 foo.py
0
0
$ python3.3 foo.py
File "foo.py", line 4
print(i) # Indentazione con tab
^
TabError: inconsistent use of tabs and spaces in indentation
```

Porting automatico da Python 2 a Python 3

La via più semplice per fare il *porting* del codice da Python 2 a Python 3 è quella di utilizzare il programma `python 2to3`. Questo fa parte della distribuzione standard di Python ed è un semplice script che utilizza il modulo `lib2to3`:

```
$ which 2to3
/usr/local/bin/2to3
$ more /usr/local/bin/2to3
#!/usr/local/bin/python3.3
import sys
from lib2to3.main import main

sys.exit(main("lib2to3.fixes"))
```

Ecco un semplice esempio di utilizzo:

```
$ echo "import sys" > mymodule.py
$ echo "print sys.platform" >> mymodule.py
$ more mymodule.py
import sys
print sys.platform
$ python2.7 mymodule.py
linux2
$ python3.3 mymodule.py
File "mymodule.py", line 2
print sys.platform
^
SyntaxError: invalid syntax
$ 2to3 -w mymodule.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored mymodule.py
--- mymodule.py (original)
+++ mymodule.py (refactored)
@@ -1,2 +1,2 @@
import sys
-print sys.platform
+print(sys.platform)
RefactoringTool: Files that were modified:
RefactoringTool: mymodule.py
marco@ubuntu:~/temp$ more mymodule.py
import sys
print(sys.platform)
$ python3.3 mymodule.py
```

Per approfondire questo argomento si può consultare la documentazione di `2to3` sul sito ufficiale: <http://docs.python.org/3/library/2to3.html>, oppure la guida che si occupa del porting da Python 2 a Python 3 e viceversa: <http://docs.python.org/3/howto/pyporting.html>. Vi è, inoltre, un ottimo libro che tratta il porting in modo molto dettagliato: <http://python3porting.com/>.

Appendice C

Il buffering dei file

Nel Capitolo 3 abbiamo parlato della funzione **built-in** `open()` e del parametro opzionale **buffering**. In questa appendice vedremo in modo dettagliato l'algoritmo che è alla base del buffering.

Nel Capitolo 3 abbiamo detto che il parametro opzionale `buffering` della funzione built-in `open()` è un numero intero utilizzato per impostare la strategia di buffering e abbiamo visto i casi `buffering=0` e `buffering=1`. Un valore del buffering maggiore di 1 definisce, invece, la *profondità del buffer*. In questo caso la politica di buffering è legata, oltre che al parametro `buffering`, anche a un attributo del file-object che definisce la risoluzione del buffer, ovvero il numero minimo di byte che possono essere trasferiti sul buffer. Questo attributo è il `CHUNK_SIZE`:

```
>>> open('myfile')._CHUNK_SIZE
8192
```

Con questa politica di buffering (`buffering = K > 1`), quindi, quando si scrive sul file-object i byte non vengono trasferiti immediatamente sul buffer, ma vengono messi in coda in attesa che il loro numero superi `CHUNK_SIZE`.

Indichiamo con **M** il numero di byte già in coda, con **N** il numero di nuovi byte che scriviamo tramite il file object e con **B** il numero di byte presenti sul buffer. Appena il numero di byte **N + M** è maggiore del `CHUNK_SIZE`, tutti gli **N + M** byte verranno trasferiti sul buffer o sul disco, a seconda del valore di **N + M**, di quello del buffering e di **B**:

- **buffering - B >= N + M**: tutti gli **N + M** byte verranno scritti sul buffer (infatti **buffering - B** è il numero di byte che mancano per riempire il buffer);
- **buffering >= N + M > buffering - B**: prima il buffer viene svuotato e dopo gli **N + M** bytes vengono scritti sul buffer;
- **buffering < N + M**: il buffer viene svuotato e gli **N + M** byte vengono scritti anch'essi sul file sottostante.

Vediamo di chiarire quanto detto, considerando, ad esempio, un file object con buffer da 9 byte:

```
>>> f = open('myfile', 'w', buffering=9)
```

Per agevolare la spiegazione, modifichiamo il `CHUNK_SIZE`, ovvero la lunghezza dei byte in coda che verranno inseriti nel buffer:

```
>>> f._CHUNK_SIZE = 2
```

Scriviamo adesso **N = 5** bytes. Poiché **M = 0** e **B = 0**, **buffering - B > N + M = 5 > CHUNK_SIZE**, i 5 byte verranno scritti sul buffer:

```
>>> f.write('12345')
5
>>> open('myfile').read()
"
```

Scriviamo adesso altri **N = 4** byte. Poiché **B = 5**, **M = 0**, **buffering - B = 4 = N + M = 4 > CHUNK_SIZE**, 5 byte verranno scritti sul buffer:

```
>>> f.write('6789')
4
```

Il buffer, a questo punto, è pieno, per cui ogni tentativo di scrittura su di esso causerà il suo svuotamento. Adesso scriviamo altri **N = 2** byte. Poiché **M = 0**, **N + M = 2 = CHUNK_SIZE** e i 2 byte non verranno scritti sul buffer ma

messi in coda, il buffer non verrà ancora svuotato sul file sottostante:

```
>>> f.write('ab')
2
>>> open('myfile').read()
"
```

Scriviamo un altro byte. Ora $M = 2$ e $N = 1$, per cui $N + M = 3 > \text{CHUNK_SIZE}$. Si ha, inoltre, $\text{buffering} > N + M > \text{buffering} - B = 0$, per cui il buffer viene svuotato e i nuovi $N + M$ byte (abc) vengono scritti sul buffer:

```
>>> f.write('c')
1
>>> open('myfile').read()
'123456789'
```

I conti vanno fatti in byte e non in numero di caratteri. Per mostrare ciò, consideriamo il seguente esempio:

```
>>> f = open('myfile', 'w', buffering=2)
>>> f._CHUNK_SIZE = 1
```

Abbiamo $M = 0$ e il buffer vuoto. Scriviamo un carattere di 2 byte, quindi $N = 2$, $M = 0$ e $\text{buffering} - B = 2 = N + M = 2 > \text{CHUNK_SIZE}$; il carattere verrà scritto sul buffer, il quale, potendo ospitare 2 soli byte, risulterà pieno:

```
>>> f.write('è') # Scrivo 2 byte
1
>>> open('myfile').read()
"
```

Scrivo altri $N = 2$ byte. Si ha $M = 0$, $B = 2$ e $\text{buffering} > N + M = 2 > \text{buffering} - B = 0$, per cui prima il buffer viene svuotato e dopo i 2 nuovi byte (è) verranno scritti su di esso:

```
>>> f.write('è')
1
>>> open('myfile').read()
'è'
```

Abbiamo ancora il buffer pieno ma la coda vuota. Scriviamo, quindi, un nuovo byte. Questo verrà messo in coda poiché $N + M = 1 = \text{CHUNK_SIZE}$:

```
>>> f.write('b')
1
>>> open('myfile').read()
'b'
```

Se adesso scriviamo un nuovo carattere da 2 byte, poiché $M = 1$, $N + M = 3 > \text{CHUNK_SIZE}$ e $\text{buffering} = 2 < M + N > \text{buffering} - 2 = 0$, il buffer viene svuotato e i 3 nuovi byte, non potendo essere scritti su un buffer di soli 2 byte, vengono direttamente scritti sul file sottostante:

```
>>> f.write('è')
1
>>> open('myfile').read()
'bèèè'
```