

- Al Sweigart -

Automatizzare le cose noiose con

# PYTHON

Programmazione pratica per principianti assoluti



[Le basi di Python >>](#)

[Leggere, scrivere, organizzare i file >>](#)

[Lavorare con Excel, i PDF, i dati JSON >>](#)

[Inviare email, modificare immagini >>](#)

\*pro  
**DigitalLifeStyle**

- Al Sweigart -

# Automatizzare le cose noiose con **PYTHON**

**Programmazione pratica per principianti assoluti**



[Le basi di Python >>](#)

[Leggere, scrivere, organizzare i file >>](#)

[Lavorare con Excel, i PDF, i dati JSON >>](#)

[Inviare email, modificare immagini >>](#)

**\*pro**  
**DigitalLifeStyle**



# Automatizzare le cose noiose con PYTHON

## Programmazione pratica per principianti assoluti

Con questo libro imparerete a usare Python per scrivere programmi che facciano in pochi minuti quello che a mano vi costerebbe ore – e non è necessario che abbiate già esperienza di programmazione. Una volta acquisite le basi, potrete creare programmi Python che svolgono senza fatica e in modo splendido operazioni ripetitive in modo completamente automatico. Istruzioni passo passo analizzeranno in dettaglio ciascun programma e progetti pratici alla fine di ciascun capitolo vi sfideranno a migliorare quei programmi e a usare le competenze appena acquisite per automatizzare operazioni simili. Non passate il vostro tempo a fare quel che potrebbe fare una scimmia ammaestrata. Anche se non avete mai scritto una riga di codice, potete fare in modo che sia il vostro computer a gestire la parte noiosa. Imparate ad *Automatizzare le cose noiose con Python!*

### Tra gli argomenti trattati

- Cercare un testo in uno o più file
- Creare, aggiornare, spostare e rinominare file e cartelle
- Fare ricerche nel Web e scaricare contenuti online
- Aggiornare e formattare dati in fogli di calcolo Excel di qualsiasi dimensione
- Suddividere o unire PDF, dotarli di watermark e cifrarli
- Inviare email di sollecito e notifiche testuali
- Elaborare serie di immagini
- Compilare moduli online

### L'autore

**Al Sweigart.** È uno sviluppatore di software e insegna a programmare a bambini e adulti. Ha scritto vari libri introduttivi a Python, tra cui *Hacking Secret Ciphers with Python*, *Invent Your Own Computer Games with Python* e *Making Games with Python & Pygame*.

\*pro  
DigitalLifeStyle

# **Automatizzare le cose noiose con Python**

## **Programmazione pratica per principianti assoluti**

**Al Sweigart**

EDIZIONI  
**LSWR**

Titolo originale: *Automate the Boring Stuff with Python*

ISBN: 978-1-59327-599-0

Published by No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

[www.nostarch.com](http://www.nostarch.com)

Cover Illustration: Josh Ellingson

Copyright © 2015 by Al Sweigart. All rights reserved.

**Autore:** Al Sweigart

**Collana:** \*pro  
DigitalLifeStyle

**Edizione italiana:**

*Automatizzare le cose noiose con Python*

**Editor in Chief:** Marco Aleotti

**Progetto grafico:** Roberta Venturieri

**Traduzione:** Virginio B. Sala

© 2017 Edizioni Lswr\* – Tutti i diritti riservati

**ISBN:** 978-88-6895-455-0

*I diritti di traduzione, di memorizzazione elettronica, di riproduzione e adattamento totale o parziale con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche), sono riservati per tutti i Paesi. Le fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941 n. 633.*

*Le fotocopie effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEARED, Centro Licenze e Autorizzazioni per le Riproduzioni Editoriali, Corso di Porta Romana 108, 20122 Milano, e-mail [autorizzazioni@clearedi.org](mailto:autorizzazioni@clearedi.org) e sito web [www.clearedi.org](http://www.clearedi.org).*

*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.*

*L'Editore ha compiuto ogni sforzo per ottenere e citare le fonti esatte delle illustrazioni. Qualora in qualche caso non fosse riuscito a reperire gli aventi diritto è a disposizione per rimediare a eventuali involontarie omissioni o errori nei riferimenti citati.*

*Tutti i marchi registrati citati appartengono ai legittimi proprietari.*

**EDIZIONI  
LSWR**

Via G. Spadolini, 7

20141 Milano (MI)

Tel. 02 881841

[www.edizionilswr.it](http://www.edizionilswr.it)

(\*) Edizioni Lswr è un marchio di La Tribuna Srl. La Tribuna Srl fa parte di **LSWR GROUP**.

# Sommario

## RINGRAZIAMENTI

## INTRODUZIONE

Per chi è questo libro?

Convenzioni

Che cos'è la programmazione?

A proposito di questo libro

Porre domande di programmazione intelligenti

Riepilogo

## PARTE 1: ELEMENTI FONDAMENTALI DELLA PROGRAMMAZIONE IN PYTHON

### 1. LE BASI DI PYTHON

Inserire espressioni nella shell interattiva

Tipi di dati: interi, virgola mobile e stringa

Concatenazione e replica di stringhe

Memorizzare valori in variabili

Il primo programma

Analisi del programma

Riepilogo

Un po' di pratica

### 2. CONTROLLO DEL FLUSSO

Valori Booleani

Operatori di confronto

Operatori Booleani

Elementi del controllo del flusso

Esecuzione del programma

Enunciati di controllo del flusso

Importare moduli

Conclusione prematura di un programma con sys.exit()

Riepilogo

Un po' di pratica

### 3. FUNZIONI

Enunciati def con parametri

Valori di ritorno ed enunciati return

Il valore None

Argomenti parola chiave e print()  
Ambito locale e globale  
Gestione delle eccezioni  
Un breve programma: Indovina il numero  
Riepilogo  
Domande di ripasso  
Un po' di pratica

## 4. LISTE

Il tipo di dati lista  
Ottenere i singoli valori in una lista mediante gli indici  
Lavorare con le liste  
Operatore di assegnazione aumentata  
Metodi  
Un programma di esempio: la palla magica con una lista  
Tipi simili a liste: stringhe e tuple  
Riferimenti  
Riepilogo  
Un po' di pratica

## 5. DIZIONARI E STRUTTURAZIONE DEI DATI

Il tipo di dati dizionario  
Una stampa elegante (pretty printing)  
Uso di strutture di dati come modelli di cose reali  
Riepilogo  
Domande di ripasso  
Un po' di pratica

## 6. MANIPOLARE STRINGHE

Lavorare con le stringhe  
Metodi utili per le stringhe  
Progetto: password locker  
Progetto: aggiungere punti elenco al markup wiki  
Riepilogo  
Domande di ripasso  
Un po' di pratica

# PARTE 2: AUTOMATIZZARE ATTIVITÀ

## 7. TROVARE CORRISPONDENZE CON LE ESPRESSIONI REGOLARI

Trovare schemi di testo senza le espressioni regolari  
Trovare schemi di testo con le espressioni regolari  
Ancora corrispondenze con le espressioni regolari  
Trovare un numero specifico di ripetizioni con le parentesi graffe

Corrispondenze avide e non  
Il metodo findall()  
Classi di caratteri  
Creare le proprie classi di caratteri  
I caratteri accento circonflesso e dollaro  
Il carattere jolly  
Riepilogo dei simboli delle espressioni regolari  
Corrispondenza case-insensitive  
Sostituire stringhe con il metodo sub()  
Gestire espressioni regolari complesse  
Combinare re.IGNORECASE, re.DOTALL e re.VERBOSE  
Progetto: estrarre numeri telefonici e indirizzi email  
Riepilogo  
Domande di ripasso  
Un po' di pratica

## 8. LEGGERE E SCRIVERE FILE

File e percorsi di file  
Il modulo os.path  
Il processo di lettura e scrittura di file  
Salvare variabili con il modulo shelve  
Salvare variabili con la funzione pprint.pformat()  
Progetto: generazione di file per test casuali  
Progetto: Appunti multipli  
Riepilogo  
Domande di ripasso  
Un po' di pratica

## 9. ORGANIZZARE I FILE

Il modulo shutil  
Percorrere un albero di directory  
Progetto: date da formato americano a europeo  
Progetto: backup di una cartella in un file ZIP  
Riepilogo  
Domande di ripasso  
Un po' di pratica

## 10. DEBUG

Eccezioni  
Ottenere il traceback come stringa  
Asserzioni  
Logging  
Debugger  
Riepilogo  
Domande di ripasso

Un po' di pratica

## 11. WEB SCRAPING

Progetto: mapIt.py con il modulo webbrowser

Caricare file dal Web con il modulo requests

HTML

Analizzare HTML con il modulo BeautifulSoup

Progetto: Ricerca in Google del tipo “mi sento fortunato”

Progetto: scaricare tutti i fumetti XKCD

Controllare il browser con il modulo selenium

Riepilogo

Domande di ripasso

Un po' di pratica

## 12. LAVORARE CON FOGLI DI CALCOLO EXCEL

Documenti Excel

Installazione del modulo openpyxl

Leggere documenti Excel

Progetto: lettura di dati da un foglio di calcolo

Scrivere documenti Excel

Progetto: aggiornamento di un foglio di calcolo

Impostare lo stile dei caratteri delle celle

Oggetti Font

Formule

Regolare righe e colonne

Grafici

Riepilogo

Domande di ripasso

Un po' di pratica

## 13. LAVORARE CON DOCUMENTI PDF E WORD

Documenti PDF

Progetto: combinare pagine selezionate da più PDF

Documenti Word

Riepilogo

Domande di ripasso

Un po' di pratica

## 14. LAVORARE CON FILE CSV E DATI JSON

Il modulo csv

Progetto: eliminare l'intestazione da file CSV

JSON e API

Progetto: estrarre dati meteorologici

Riepilogo

Domande di ripasso

Un po' di pratica

## 15. TENERE TRACCIA DELL'ORA, PIANIFICARE ATTIVITÀ E LANCIARE PROGRAMMI

Il modulo time

Arrotondare numeri

Progetto: supercronometro

Il modulo datetime

Riepilogo delle funzioni di data e ora di Python

Progetto: scaricamento multithread da XKCD

Lanciare altri programmi da Python

Progetto: un semplice programma per il conteggio alla rovescia

Riepilogo

Domande di ripasso

Un po' di pratica

## 16. INVIARE EMAIL E SMS

SMTP

IMAP

Recuperare ed eliminare messaggi con IMAP

Progetto: inviare email di sollecito ai membri di un club

Inviare sms con Twilio

Progetto: un modulo “mandami solo un sms”

Riepilogo

Domande di ripasso

Un po' di pratica

## 17. MANIPOLARE IMMAGINI

Nozioni fondamentali sulle immagini

Manipolare immagini con Pillow

Progetto: aggiungere un logo

Disegnare su immagini

Riepilogo

Domande di ripasso

Un po' di pratica

## 18. CONTROLLARE TASTIERA E MOUSE CON L'AUTOMAZIONE DELLA GUI

Installazione del modulo pyautogui

Come rimanere sulla strada giusta

Controllare i movimenti del mouse

Progetto: “Dov’è il mouse proprio adesso?”

Controllare le interazioni del mouse

Lavorare con lo schermo

Progetto: estendere il programma mouseNow

Riconoscimento di immagini

Controllare la tastiera

Panoramica delle funzioni di PyAutoGUI

Progetto: compilatore automatico di moduli

Riepilogo

Domande di ripasso

Un po' di pratica

## APPENDICE A: INSTALLARE MODULI DI TERZE PARTI

Lo strumento pip

Installare moduli di terze parti

## APPENDICE B: ESEGUIRE I PROGRAMMI

La riga shebang

Eseguire programmi Python in Windows

Eseguire programmi Python in OS X e Linux

Eseguire programmi Python con le asserzioni disattivate

## APPENDICE C: RISPOSTE ALLE DOMANDE DI RIPASSO

# Ringraziamenti

Non avrei potuto scrivere un libro come questo senza l'aiuto di un gran numero di persone. Vorrei ringraziare Bill Pollock; i miei editor Laurel Chun, Leslie Shen, Greg Poulos e Jennifer Griffith-Delgado, e tutto il resto dello staff della No Starch Press per il loro prezioso aiuto. Grazie al revisore tecnico, Ari Lacenski, per gli ottimi suggerimenti, la redazione e il sostegno.

Molti ringraziamenti al nostro Benevolo Dittatore a Vita, Guido van Rossum, e a tutti alla Python Software Foundation per il loro grande lavoro. La community di Python è la migliore che abbia trovato in tutto il settore tecnologico.

Infine, vorrei ringraziare la mia famiglia, gli amici e la compagnia da Shotwell's per non essersela presa per la scarsa vita sociale che ho avuto mentre scrivevo questo libro.

Al Sweigart

# Introduzione

“Hai appena fatto in due ore quello che a noi richiede due giorni di lavoro.” Il mio compagno di stanza al college lavorava, agli inizi degli anni 2000, in un negozio di elettronica. Ogni tanto il negozio riceveva un foglio di calcolo con migliaia di prezzi di prodotti della concorrenza e una squadra di tre dipendenti lo stampava su un pacco di carta, poi si divideva il malloppo e, per ciascun prezzo, i tre consultavano il prezzo praticato in negozio e annotavano tutti i prodotti che i concorrenti vendevano a meno. Di solito tutta l’operazione richiedeva un paio di giorni.

“Sapete, potrei scrivere un programma che faccia tutto questo, se avete il file originale da cui avete ottenuto le stampate”, ha detto loro il mio compagno di stanza, quando li ha visti seduti sul pavimento con i fogli sparsi e ammucchiati intorno a loro.

Dopo un paio d’ore, aveva un programmino che leggeva da un file il prezzo praticato da un concorrente, trovava il prodotto corrispondente nel database del negozio e annotava se il prezzo del concorrente era minore. Non programmava da molto tempo, e doveva ancora perdere una gran quantità di tempo a consultare la documentazione in un libro di programmazione. L’esecuzione del programma richiedeva solo pochi secondi. Il mio compagno di stanza e i suoi colleghi quel giorno si sono concessi una pausa pranzo straordinariamente lunga.

Questo è il potere della programmazione. Un computer è come un coltellino dell’esercito svizzero, che si può configurare per un’infinità di cose. Molti passano ore a fare clic e a scrivere sulla tastiera per svolgere compiti ripetitivi, senza rendersi conto che la macchina che stanno usando potrebbe fare il loro lavoro in pochi secondi, se solo le impartissero le istruzioni giuste.

## Per chi è questo libro?

Il software è alla base di così tanti strumenti che usiamo oggi: quasi chiunque usa i social network per comunicare, molti hanno nei loro telefoni computer collegati a Internet, e la maggior parte delle mansioni d’ufficio comportano interagire con un computer per svolgere il proprio lavoro. Così, la domanda di persone in grado di scrivere codice è salita alle stesse. Un numero incredibile di libri, di tutorial interattivi sul Web e di seminari per sviluppatori promette di trasformare gli ambiziosi alle prime armi in ingegneri del software con stipendi a sei cifre.

Questo libro non è per loro. È per tutti gli altri.

Da solo, questo libro non vi trasformerà in uno sviluppatore di software professionale, così come poche lezioni di chitarra non vi trasformeranno in una rockstar. Ma, se lavorate in un ufficio, se siete amministratori, accademici o comunque persone che usano un computer per lavoro o per divertimento, imparerete le basi della programmazione, e potrete automatizzare compiti semplici come questi:

- spostare e cambiare nome a migliaia di file e organizzarli in cartelle;
- compilare moduli online senza dover scrivere alla tastiera;
- scaricare file o copiare testi da un sito web ogni volta che viene aggiornato;
- fare in modo che il vostro computer vi mandi notifiche personalizzate;
- aggiornare o formattare fogli di calcolo Excel;
- controllare la posta elettronica e spedire risposte preconfezionate.

Questi compiti sono semplici, ma agli esseri umani richiedono una gran quantità di tempo, ma spesso sono così banali o specifici che non esiste software già pronto per svolgerli. Armati con un po' di conoscenza della programmazione, potrete fare in modo che sia il vostro computer a svolgerli per voi.

## Convenzioni

Questo libro non è organizzato come un manuale di consultazione: è una guida per chi è alle prime armi. Lo stile di codifica ogni tanto non è in linea con le pratiche migliori (per esempio, qualche programma usa variabili globali), ma è un compromesso per rendere il codice più facile da imparare. Il libro è pensato per persone che scrivono codice usa e getta, perciò non dedica molto tempo allo stile e all'eleganza. Concetti di programmazione sofisticati – come la programmazione a oggetti, le comprensioni di lista e i generatori – non sono trattati, per la complessità che aggiungono. Programmatori esperti potranno indicare modi in cui il codice in questo libro potrebbe essere modificato per migliorarne l'efficienza, ma l'intento è soprattutto quello di avere dei programmi con cui lavorare con il minimo sforzo.

## Che cos'è la programmazione?

Spettacoli televisivi e film spesso mostrano programmati che scrivono furiosamente serie misteriose di 1 e 0 su schermi fosforescenti, ma la programmazione moderna non è così misteriosa. Programmare è semplicemente l'inserimento di istruzioni che il computer deve eseguire. Queste istruzioni possono macinare numeri, modificare testo, cercare informazioni in file, o comunicare con altri computer via Internet.

Tutti i programmi usano come “mattoni da costruzione” alcune istruzioni di base. Ecco alcune delle più comuni, in italiano:

- “Fai questo; poi fai quest’altro.”
- “Se questa condizione è vera, compi questa azione; altrimenti, compi quest’altra azione.”
- “Esegui questa azione quel numero di volte.”
- “Continua a fare questo finché questa condizione non diventa vera.”

Possiamo combinare questi mattoni per realizzare decisioni più complicate. Per esempio, queste sono le istruzioni di programmazione, il codice sorgente, di un semplice programma scritto nel linguaggio di programmazione Python. Partendo dall'inizio, il software Python esegue ciascuna riga del codice (alcune vengono eseguite solo se una certa condizione è vera, altrimenti Python esegue qualche altra riga), fino a che non arriva in fondo.

```
❶ passwordFile = open('SecretPasswordFile.txt')
❷ secretPassword = passwordFile.read()
❸ print('Scrivi la tua password.')
typedPassword = input()
❹ if typedPassword == secretPassword:
    ❺     print('Accesso consentito')
    ❻     if typedPassword == '12345':
            ❼         print('Questa è un password che solo un idiota mette alla sua valigia.')
    ❽ else:
        ❾         print('Accesso negato')
```

Non saprete magari nulla di programmazione, ma (se sapete qualche parola di inglese) probabilmente potrete indovinare ragionevolmente che cosa fa il codice precedente, semplicemente leggendolo. In primo luogo viene aperto il file *SecretPasswordFile.txt* ①, e viene letta la password segreta che contiene ②. Poi, all'utente viene chiesto di inserire una password (dalla tastiera) ③. Le due password vengono confrontate ④ e, se sono identiche, il programma stampa sullo schermo Accesso consentito ⑤. Poi il programma controlla se la password è 12345 ⑥ ed eventualmente fa notare che non si tratta della scelta migliore di una password ⑦. Se le password non sono identiche, il programma invece stampa sullo schermo Accesso negato ⑧.

## Che cos'è Python?

Python indica il linguaggio di programmazione Python (con regole sintattiche che definiscono quello che è considerato codice Python valido) e anche il software interprete Python, che legge codice sorgente (scritto nel linguaggio Python) e ne esegue le istruzioni. L'interprete Python può essere scaricato liberamente da <http://python.org/> e ne esistono versioni per Linux, OS X e Windows.

Il nome Python arriva dal gruppo di comici inglesi **Monty Python**, non dal **serpente** (in inglese, python = pitone). Chi programma in Python viene chiamato affettuosamente un pythonista e capita spesso che i tutorial e la documentazione di Python siano pieni di riferimenti sia ai Monty Python che ai serpenti.

## Chi programma non ha bisogno di sapere molto di matematica

La preoccupazione che molti manifestano, all'idea di imparare a programmare, è che si debba conoscere molta matematica. In realtà, la maggior parte della programmazione non richiede altro che l'aritmetica elementare. Essere bravi nella programmazione non è molto diverso dall'essere bravi nel risolvere Sudoku.

Per risolvere un Sudoku, bisogna disporre i numeri da 1 a 9 in ogni riga, ogni colonna e ogni riquadro  $3 \times 3$  del rettangolo  $9 \times 9$  completo. La soluzione si trova mediante deduzione e logica a partire dai numeri già scritti. Per esempio, poiché il 5 si trova nella casella superiore sinistra del Sudoku nella Figura I.1, non può più comparire in alcuna casella prima riga, della colonna più a sinistra o nel quadrato  $3 \times 3$  in alto a sinistra. Risolvendo una riga, una colonna o un quadrato alla volta si ottengono più indizi numerici per completare il resto del rompicapo.

5	3		7					
6			1	9	5			
	9	8				6		
8			6				3	
4		8	3			1		
7		2				6		
	6			2	8			
		4	1	9			5	
			8		7	9		

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

**Figura I.1** - Un rompicapo Sudoku (a sinistra) e la sua soluzione (a destra). Anche se si usano i numeri, il Sudoku non richiede molte conoscenze matematiche. (Immagini © Wikimedia Commons)

Solo perché nel Sudoku compaiono dei numeri non vuol dire che per trovare la soluzione sia necessario essere bravi in matematica, e lo stesso vale per la programmazione.

Come accade quando si risolve un Sudoku, quando si scrive un programma bisogna suddividere un problema in singoli passi dettagliati. Analogamente, quando si effettua il debug dei programmi (cioè si cerca di identificare e correggere gli errori), bisogna osservare con pazienza che cosa fa il programma e trovare la causa degli errori – i “bug”. Come accade in tutte le attività, quanto più programmerete, tanto più bravi diventerete.

## Programmare è un’attività creativa

Programmare è un’attività creativa, un po’ come costruire un castello con i mattoncini LEGO. Si parte con un’idea di massima dell’aspetto che si vuole dare al castello e con un inventario dei mattoncini disponibili. Poi si comincia a costruire. Una volta finito di costruire il vostro programma, potete abbellire il vostro codice come fareste con il castello.

La differenza fra programmare e altre attività creative è che, quando programmate, avete tutte le “materie prime” di cui avete bisogno all’interno del vostro computer: non dovete acquistare ancora tele, vernici, pellicola, filo, mattoncini LEGO o componenti elettronici. Quando avete scritto il vostro programma, potete condividerlo online senza fatica con il resto del mondo. E, anche se nel programmare commetterete degli errori, l’attività è comunque molto divertente.

## A proposito di questo libro

La prima parte di questo libro tratta i concetti fondamentali della programmazione in Python, mentre la seconda parte affronta le varie attività che potete far svolgere automaticamente al vostro computer. Ciascun capitolo della seconda parte contiene programmi che potete studiare. Ecco una veloce carrellata su quello che troverete in ciascun capitolo.

- **Capitolo 1: Le basi di Python** Tratta delle espressioni, il tipo più fondamentale di istruzione in Python, e di come usare la shell interattiva di Python per sperimentare con il codice.
- **Capitolo 2: Controllo del flusso** Spiega come far sì che siano i programmi a decidere quali istruzioni eseguire, in modo che il vostro codice possa rispondere con intelligenza a condizioni diverse.
- **Capitolo 3: Funzioni** Vi spiega come definire le vostre funzioni, per poter organizzare il codice in spezzoni gestibili con maggiore facilità.
- **Capitolo 4: Liste** Introduce il tipo di dati lista e spiega come organizzare i dati.
- **Capitolo 5: Dizionari e strutturazione dei dati** Introduce il tipo d dati dizionario e presenta modalità più potenti di organizzazione dei dati.
- **Capitolo 6: Manipolare le stringhe** Tratta di come lavorare con dati testuali (chiamati stringhe in Python).

## Parte II: Automatizzare attività

- **Capitolo 7: Trovare corrispondenze con le espressioni regolari** Tratta di come Python possa manipolare stringhe e cercare configurazioni di testo mediante le espressioni regolari.
- **Capitolo 8: Leggere e scrivere file** Spiega come i vostri programmi possono leggere i contenuti di file di testo e salvare informazioni in file sul disco fisso.
- **Capitolo 9: Organizzare i file** Mostra come Python possa copiare, spostare, rinominare e cancellare un gran numero di file molto più rapidamente di un essere umano. Spiega anche la compressione e decompressione dei file.
- **Capitolo 10: Debugging** Mostra come usare i vari strumenti di Python per la ricerca e correzione degli errori.
- **Capitolo 11: Web scraping** Mostra come scrivere programmi in grado di scaricare automaticamente pagine web e analizzare alla ricerca di informazioni. Questo è il cosiddetto web scraping.
- **Capitolo 12: Lavorare con fogli di calcolo Excel** Tratta della manipolazione di fogli di calcolo Excel da programma, in modo da non doverli nemmeno leggere. Questo è utile quando il numero dei documenti che dovete analizzare è nell'ordine delle centinaia o delle migliaia.
- **Capitolo 13: Lavorare con documenti PDF e Word** Tratta di come leggere da programma documenti Word e PDF.
- **Capitolo 14: Lavorare con file CSV e dati JSON** Continua a spiegare come manipolare da programmi documenti con file CSV e JSON.
- **Capitolo 15: Tenere traccia dell'ora, pianificare attività e lanciare programmi** Spiega come i programmi Python gestiscano ore e date e come pianificare il computer perché svolga determinate attività in momenti prestabiliti. Questo capitolo mostra anche come i vostri programmi Python possano lanciare programmi non scritti in Python.
- **Capitolo 16: Inviare email e sms** Spiega come scrivere programmi in grado di inviare email e messaggi di testo per vostro conto.
- **Capitolo 17: Manipolare immagini** Spiega come realizzare programmi in grado di manipolare immagini, per esempio file JPEG o PNG.
- **Capitolo 18: Controllare tastiera e mouse con l'automazione della GUI** Spiega come controllare da programma mouse e tastiera per automatizzare i clic e la pressione dei tasti.

## Scaricare e installare Python

Potete scaricare gratuitamente Python per Windows, OS X e Ubuntu dall'indirizzo <http://python.org/downloads/>. Se scaricate da questa pagina la versione più recente, tutti i programmi di questo libro dovrebbero funzionare perfettamente.

Fate attenzione: dovete scaricare una versione di Python 3 (per esempio, 3.4.0). I programmi di questo libro sono stati scritti per poter girare in Python 3 ed è possibile che non girino correttamente, o che non vengano affatto eseguiti, in Python 2.

Troverete programmi di installazione per computer a 64 e a 32 bit per ciascun sistema operativo, perciò per prima cosa dovete stabilire quale sia il programma di installazione che vi serve. Se avete acquistato il vostro computer nel 2007 o dopo, è molto probabile che abbiate un sistema a 64 bit. Altrimenti, avrete sicuramente una versione a 32 bit, ma ecco come scoprirlo per certo:

- In Windows, selezionate **Start > Pannello di controllo > Sistema** e verificate se il *Tipo sistema* sia a 64 o a 32 bit.
- In OS X, andate al menu **Mela**, selezionate **Su questo Mac > Altre informazioni > Sistema > Hardware**, poi esamineate il campo *Nome processore*. Se dice Intel Core Solo o Intel Core Duo, avete una macchina a 32 bit. Se dice qualcos'altro (anche Intel Core 2 Duo), avete una macchina a 64 bit.
- In Ubuntu Linux, aprite una finestra di Terminale e quindi impartite il comando `uname -m`. Se la risposta è `i686` il sistema è a 32 bit, se la risposta è `x86_64` il sistema è a 64 bit.

In Windows, scaricate il programma di installazione di Python (il nome di file avrà come suffisso `.msi`) e fate un doppio clic su di esso. Per installare Python, seguite le istruzioni che il programma mostra sullo schermo:

1. Selezionate Installa per tutti gli utenti, poi fate clic su Avanti.
2. Installate nella cartella `C:\Python34` (il numero varierà a seconda della versione di Python che installate) facendo clic su Avanti.
3. Fate clic nuovamente su Avanti per saltare la sezione di personalizzazione.

Sotto Mac OS X, scaricate il file `.dmg` giusto per la vostra versione del sistema operativo e fate su di esso un doppio clic. Per installare Python, seguite le istruzioni che il programma mostra sullo schermo:

1. Quando il pacchetto DMG si apre in una nuova finestra, fate doppio clic sul file `Python.mpkg`. È possibile che dobbiate inserire la password da amministratore.
2. Fate clic su *Continua* per passare la sezione di benvenuto e fate clic su Accetto per accettare l'accordo di licenza.
3. Selezionate *HD Macintosh* (o il nome del vostro disco fisso) e fate clic su *Installa*.

Sotto Ubuntu, potete installare Python dal Terminale seguendo questi passi:

1. Aprite la finestra del Terminale.
2. Inserite `sudo apt-get install python3`.
3. Inserite `sudo apt-get install idle 3`.
4. Inserite `sudo apt-get install python3-pip`.

Mentre l'interprete Python è il software che esegue I vostri programmi, il software IDLE (*Interactive DeveLopment Environment*, ovvero “ambiente di sviluppo interattivo”) è l’ambiente in cui scriverete i vostri programmi, un po’ come un elaboratore di testo. Avviamo ora IDLE.

- In Windows 7 o versioni successive, fate clic sull’icona *Start* nell’angolo inferiore sinistro dello schermo, scrivete IDLE nella casella di ricerca, poi selezionate **IDLE (Python GUI)**.
- In Windows XP, fate clic sul pulsante Start, poi selezionate **Programmi > Python 3.4 > IDLE (Python GUI)**.
- In Mac OS X, aprite la finestra del Finder, fate clic su **Applicazioni**, fate clic su **Python 3.4** e infine fate clic sull’icona di IDLE.
- In Ubuntu, selezionate **Applicazioni > Accessori > Terminale** e quindi inserite `idle3`. (Potete anche fare clic su **Applicazioni** nella parte superiore dello schermo, selezionare **Programmazione** e poi fare clic su **IDLE3**).

## La shell interattiva

Indipendentemente da quale sia il vostro sistema operativo, la finestra di IDLE che si aprirà si presenterà quasi del tutto vuota, tranne per poche righe di testo che saranno simili a queste (il numero della versione dipenderà da quale versione avete installato)::

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license( )" for more information.
>>>
```

Questa finestra è chiamata **shell interattiva**. Una shell è un programma che permette l’inserimento di istruzioni per il computer, in modo non tanto diverso da Terminale o Prompt dei comandi in OS X e Windows, rispettivamente. La shell interattiva di Python vi consente di scrivere istruzioni che l’interprete Python dovrà eseguire. Il computer legge le istruzioni che inserite e le esegue immediatamente.

Per esempio, inserite questa istruzione nella shell interattiva, dopo il prompt `>>>`:

```
>>> print('Ciao mondo!')
```

Quando avrete scritto questa riga di testo e avrete premuto Invio, la shell interattiva vi risponderà in questo modo:

```
>>> print('Ciao mondo!')
Ciao mondo!
```

## Come trovare aiuto

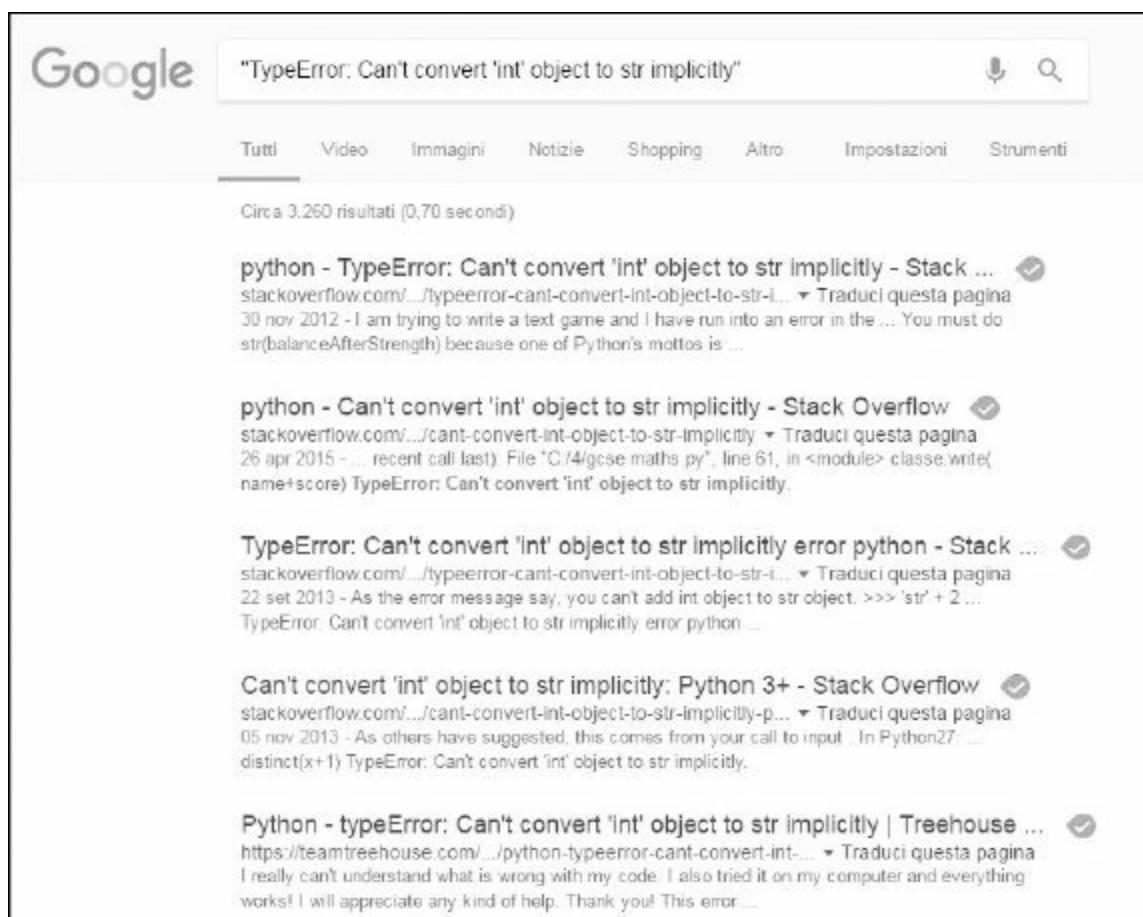
Risolvere da soli problemi di programmazione è più facile di quel che probabilmente pensate. Se non siete convinti, proviamo a provocare apposta un errore: inserite nella shell interattiva ‘`42 + 3`’. Non c’è bisogno per ora che sappiate che cosa significa questa istruzione, ma il risultato dovrebbe essere simile a questo:

```

>>> '42' + 3
❶ Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    '42' + 3
❷ TypeError: Can't convert 'int' object to str implicitly
>>>

```

Il messaggio d'errore ❷ è comparso qui perché Python non era in grado di dare un significato all'istruzione inserita. La parte traceback ❶ del messaggio d'errore mostra la specifica istruzione e il numero della riga con cui Python ha avuto problemi. Se non siete sicuri di che cosa significhi un particolare messaggio d'errore, cercate online quel preciso messaggio. Scrivete “**TypeError: Can't convert 'int' object to str implicitly**” (comprese le virgolette) nel motore di ricerca che preferite, e vedrete un gran numero di collegamenti a pagine in cui si spiega che cosa significa quel messaggio d'errore e che cosa lo provoca, come si può vedere nella [Figura I.2](#).



**Figura I.2** - I risultati forniti da Google per un messaggio d'errore possono essere molto utili. (Per vedere solo le pagine in italiano, selezionate la lingua sotto il menu Strumenti.)

Troverete spesso che qualcun altro ha avuto il vostro stesso problema e che qualche persona di buona volontà ha già fornito una risposta. Nessuno può sapere tutto sulla programmazione, perciò una parte del lavoro quotidiano di uno sviluppatore software è sempre cercare risposte a domande tecniche.

## Porre domande di programmazione intelligenti

Se non riuscite a trovare la risposta cercando online, provate a porre la domanda in un forum web come Stack Overflow (<http://stackoverflow.com/>) o la sezione “learn programming” di reddit all’indirizzo <http://reddit.com/r/learnprogramming/>.

Ricordate però che vi sono modi intelligenti di formulare domande di programmazione, che aiutano gli altri ad aiutarvi. Leggete sempre le sezioni delle Frequently Asked Questions (FAQ) di questi siti per capire quale sia il modo più corretto di porre le domande.

Nel porre domande di programmazione, tenete sempre presenti queste indicazioni:

Spiegate che cosa state cercando di fare, non solo che cosa avete fatto. Questo permette a chi vi vuole aiutare di sapere se siete sulla strada sbagliata.

Specificate il punto in cui si presenta l'errore. Si presenta proprio all'inizio del programma o solo dopo una ben precisa azione?

Copiate e incollate tutto il messaggio d'errore e il vostro codice in <http://pastebin.com> o <http://gist.github.com>. Questi siti web semplificano la condivisione di grandi quantità di codice attraverso il Web, senza il rischio di perdere formattazione del testo. Poi potete inserire l'URL del codice nella vostra email o nel post sul forum. Per esempio, qui trovate alcuni pezzi di codice che ho postato io: <http://pastebin.com/SzP2DbFx/> e [https://gist.github.com/asweigart/6912168/](https://gist.github.com/asweigart/6912168).

Spiegate che cosa avete già cercato di fare per risolvere il vostro problema. Questo dice agli altri che avete già lavorato un po' per i fatti vostri per capire che cosa non va.

Indicate la versione di Python che state usando. (Vi sono differenze fondamentali fra gli interpreti delle versioni 2 e 3 di Python.) Indicate inoltre quale sistema operativo utilizzate, e in quale versione.

Se l'errore si è verificato dopo che avete apportato un cambiamento nel vostro codice, spiegate esattamente che cosa avete modificato.

Precisate se siete in grado di riprodurre l'errore ogni volta che eseguite il programma o se si verifica solo quando compite qualche particolare azione. Spiegate in tal caso di quali azioni si tratta.

Seguite sempre anche le regole dell'etichetta nel Web. Per esempio, non scrivete le vostre domande in tutte maiuscole e non abbiate pretese eccessive nei confronti delle persone che cercano di aiutarvi.

## Riepilogo

Per la maggior parte delle persone, il computer è solo un elettrodomestico, non un utensile. Ma se imparate a programmare, avrete accesso a uno degli strumenti più potenti del mondo moderno, e lungo la strada non vi mancheranno le occasioni per divertirvi. Programmare non è come fare un intervento chirurgico al cervello: chi è alle prime armi può tranquillamente sperimentare e permettersi di commettere errori.

Mi piace aiutare le persone a scoprire Python. Scrivo tutorial di programmazione sul mio blog <http://inventwithpython.com/blog/> e potete contattarmi e pormi domande all'indirizzo [al@inventwithpython.com](mailto:al@inventwithpython.com).

Questo libro vi metterà sulla buona strada partendo da una totale assenza di conoscenze sulla programmazione, ma sicuramente avrete domande che vanno al di là dei suoi contenuti. Ricordate che porre domande efficaci e sapere come trovare risposte sono strumenti preziosi per il vostro viaggio nella programmazione.

Cominciamo!

# Parte 1

## Elementi fondamentali della programmazione in Python

# Le basi di Python

Il linguaggio di programmazione **Python** ha un gran numero di **costrutti sintattici**, di **funzioni standard** di libreria e di caratteristiche dell'**ambiente di sviluppo** interattivo. Per fortuna, potete ignorarne la maggior parte; dovete solo impararne abbastanza per scrivere qualche programma utile.

Prima di fare qualsiasi cosa, però, dovete imparare alcuni concetti di programmazione di base. Come apprendisti stregoni, potreste pensare che questi concetti siano strani e noiosi, ma con un po' di conoscenza e di pratica, riuscirete a comandare il vostro computer come una bacchetta magica per fare cose incredibili.

Questo capitolo presenta qualche esempio che dovete scrivere nella shell interattiva, che consente di eseguire le istruzioni Python una per una, vedendo istantaneamente i risultati. L'uso della shell interattiva è ottimo per imparare che cosa fanno le istruzioni fondamentali di Python, perciò provate, mentre seguite il testo. Ricorderete le cose che fate molto meglio delle cose che avete solo letto.

## Inserire espressioni nella shell interattiva

Mandate in esecuzione la shell interattiva lanciando IDLE, che avete installato con Python nell'Introduzione. In Windows, aprite il menu *Start*, selezionate **Tutti i programmi > Python 3.5**, poi selezionate **IDLE (Python GUI)**. In OS X, selezionate **Applicazioni > MacPython 3.5 > IDLE**. In Ubuntu, aprite una nuova finestra di Terminale e scrivete `idle3`.

Si visualizzerà una finestra con il prompt `>>>`: quella è la **shell interattiva**. Inserite `2 + 2` al prompt, per far fare un po' di semplice matematica a Python.

```
>>> 2 + 2  
4
```

La finestra IDLE a questo punto presenterà questo testo:

```
>>> 2 + 2
```

```
4
```

```
>>>
```

In Python,  $2 + 2$  è una **espressione**, che è la forma più elementare di istruzione di programmazione del linguaggio. Le espressioni sono costituite da *valori* (come 2) e da *operatori* (come +) e possono sempre *essere valutate* (si dice anche **valorizzate**), cioè ridursi a un singolo valore. Questo significa che si può usare espressioni in qualunque punto del codice Python in cui si possa usare anche un valore.

Nell'esempio precedente,  $2 + 2$  viene valutato a un singolo valore, 4. Anche un singolo valore senza operatori è considerato un'espressione, anche se la sua valutazione dà ancora quel medesimo valore, come in questo caso:

```
>>> 2
```

```
2
```

### Gli errori vanno bene!

I programmi "vanno in crash", ovvero si bloccano, se contengono codice che il computer non comprende, e questo fa sì che Python mostri un messaggi d'errore. Un messaggio d'errore non rovinerà il vostro computer, perciò non abbiate timore di commettere errori. Un *crash* significa solamente che il programma smette di funzionare in modo imprevisto.

Se volete sapere di più su un messaggio d'errore, potete cercare l'esatto testo del messaggio online, per trovare ulteriori informazioni. Potete cercare fra le risorse all'indirizzo <http://hostarch.com/automatestuff/> per vedere un elenco dei messaggi d'errore più comuni di Python, con il loro significato (il sito è in inglese).

Esistono molti altri operatori che potete utilizzare nelle espressioni Python. La [Tabella 1.1](#) elenca tutti gli operatori matematici del linguaggio.

**Tabella 1.1** - Operatori matematici, in ordine di precedenza decrescente.

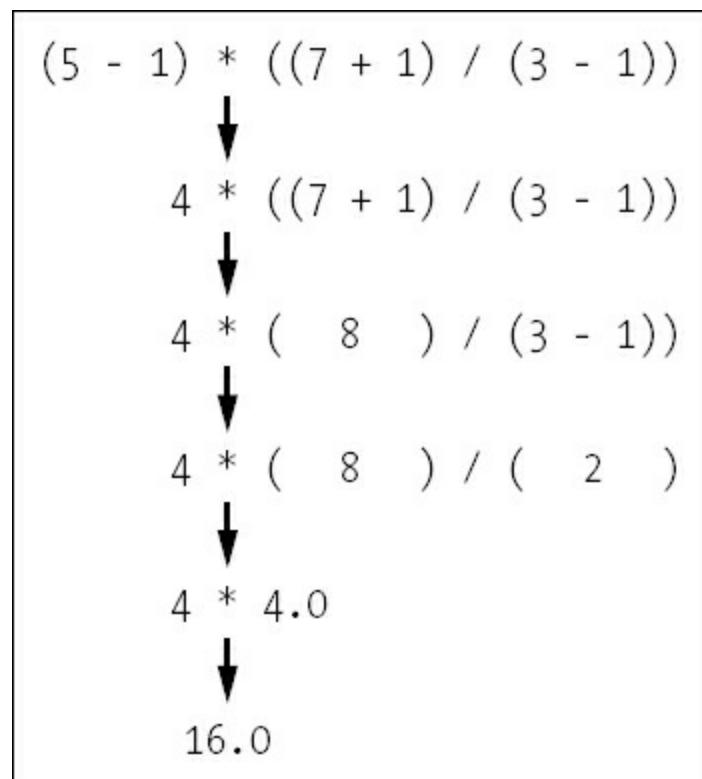
Operatore	Operazione	Esempio	Dà come valore...
<code>**</code>	Esponente	<code>2 ** 3</code>	8
<code>%</code>	Modulo/resto	<code>22 % 8</code>	6
<code>//</code>	Divisione intera/ quoziente senza resto	<code>22 // 8</code>	2
<code>/</code>	Divisione	<code>22 / 8</code>	2.75
<code>*</code>	Moltiplicazione	<code>3 * 5</code>	15
<code>-</code>	Sottrazione	<code>5 - 2</code>	3
<code>+</code>	Addizione	<code>2 + 2</code>	4

L'ordine in cui vengono eseguite le operazioni matematiche in Python (**l'ordine di precedenza degli operatori**) è simile a quello della matematica standard. Per primo viene valutato l'operatore `**`; poi gli operatori `*`, `/`, `//` e `%`, da sinistra a destra; gli operatori `+` e `-` sono valutati per ultimi (sempre da sinistra verso destra). Se necessario, potete usare le **parentesi** per aggirare il normale ordine di

precedenza. Provate a inserire queste espressioni nella shell interattiva:

```
>>> 2 + 3 * 6  
20  
>>> (2 + 3) * 6  
30  
>>> 48565878 * 578453  
28093077826734  
>>> 2 ** 8  
256  
>>> 23 / 7  
3.2857142857142856  
>>> 23 // 7  
3  
>>> 23 % 7  
2  
>>> 2 + 2  
4  
>>> (5 - 1) * ((7 + 1) / (3 - 1))  
16.0
```

In ciascun caso, come programmati dovete inserire l'espressione, ma Python fa il lavoro duro di valutarla, dandovi un valore singolo. Python continuerà a valutare le varie parti dell'espressione fino a che non arriva a un singolo valore, come si vede nella [Figura 1.1](#).



**Figura 1.1** - La valutazione di un'espressione la riduce a un singolo valore.

Queste regole per comporre operatori e valori a formare espressioni sono una parte fondamentale di Python come linguaggio di programmazione, esattamente come le **regole grammaticali** che ci permettono di comunicare. Per esempio:

Questa è una frase italiana grammaticalmente corretta.  
Questa corretta una è grammaticalmente italiana frase.

La seconda riga è difficile da analizzare, perché non segue le regole grammaticali della lingua italiana. Analogamente, se scrivete un'istruzione Python scorretta, Python non sarà in grado di comprenderla e visualizzerà un **messaggio d'errore** `SyntaxError`, come in questo caso:

```
>>> 5 +
File "<stdin>", line 1
5 +
^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
File "<stdin>", line 1
42 + 5 + * 2
^
SyntaxError: invalid syntax
```

Potete sempre verificare se un'istruzione funziona, scrivendola nella shell interattiva. Non preoccupatevi: non potete rompere il computer, la cosa peggiore che possa succedere è che Python risponda con un messaggio d'errore. Chi sviluppa software professionalmente incontra continuamente messaggi d'errore mentre scrive il proprio codice.

## Tipi di dati: interi, virgola mobile e stringa

Ricordate che le espressioni sono semplicemente valori combinati con operatori e che vengono sempre valutate, dando un valore singolo. Un **tipo di dati** è una categoria di valori, e ogni valore appartiene esattamente a un tipo di dati. I tipi di dati più comuni in Python sono elencati nella [Tabella 1.2](#). I valori -2 e 30, per esempio, si dice che sono valori *interi*. Il tipo di dati intero (o *int*) indica valori che sono numeri interi. I numeri con un punto decimale (in Python si usa il punto, all'uso inglese, non la virgola), come 3.14, sono chiamati **numeri in virgola mobile** (o *float*). Notate che, anche se il valore 42 è un intero, il valore 42.0 è un numero in virgola mobile.

**Tabella 1.2** - Tipi di dati comuni.

Tipo di dati	Esempi
Interi	-2, -1, 0, 1, 2, 3, 4, 5
Numeri in virgola mobile	-1.25, -1.0, -0.5, 0.0, 0.5, 1.0, 1.25
Stringhe	'a', 'aa', 'aaa', 'Ciao!', '11 gatti'

I programmi Python possono avere anche valori di testo, che sono chiamati **stringhe** (o *strs*, da pronunciare “stirs”). Le stringhe devono sempre essere racchiuse tra **apici singoli** ('), come ‘Ciao’ o ‘Addio mondo crudele!’, in modo che Python sappia dove la stringa inizia e dove finisce. È possibile avere anche una stringa senza alcun carattere, ‘ ‘, che è chiamata **stringa vuota**. Delle stringhe parleremo più approfonditamente nel [Capitolo 4](#).

Se vi capita di incontrare un messaggio d'errore `SyntaxError: EOL while scanning string literal`, con ogni probabilità vi siete dimenticati l'apice finale di chiusura della stringa, come in questo esempio:

```
>>> 'Ciao mondo!
SyntaxError: EOL while scanning string literal
```

## Concatenazione e replica di stringhe

Il significato di un operatore può cambiare, in funzione dei tipi di dati dei valori a cui è applicato. Per esempio, + è l'operatore di addizione quando è applicato a due interi o a due valori in virgola mobile. Quando invece è applicato a due valori stringa, è l'operatore di **concatenazione di stringhe**, accoda cioè la seconda stringa alla prima.

Provate a scrivere nella shell interattiva:

```
>>> 'Alice' + 'Bob'  
'AliceBob'
```

L'espressione viene valutata e dà un singolo valore stringa che combina il testo delle due stringhe di partenza. Se però tentate di usare l'operatore + su una stringa e un valore intero, Python non saprà come comportarsi, e visualizzerà di conseguenza un messaggio d'errore.

```
>>> 'Alice' + 42  
Traceback (most recent call last):  
File "<pyshell#26>", line 1, in <module>  
'Alice' + 42  
TypeError: Can't convert 'int' object to str implicitly
```

Il messaggio d'errore Can't convert 'int' object to str implicitly significa che Python ha pensato che voleste cercare di concatenare un intero alla stringa 'Alice'. Il vostro codice deve convertire esplicitamente l'intero in una stringa, perché Python non è in grado di farlo automaticamente. (Delle conversioni fra tipi di dati parleremo in "[Analisi del programma](#)" a pagina 13 quando parleremo delle funzioni str(), int() e float().)

L'operatore \* è utilizzato per la moltiplicazione, quando è applicato a due interi o a due valori in virgola mobile, ma quando è applicato a un valore stringa e un intero, diventa l'operatore di **replica di stringa**. Inserite una stringa moltiplicata per un numero nella shell interattiva, per vedere che cosa succede.

```
>>> 'Alice' * 5  
'AliceAliceAliceAlice'
```

L'espressione viene valutata: il risultato è un valore stringa singolo, che è la ripetizione della stringa originale per un numero di volte pari al valore intero. La replica di stringhe è un trucco utile, ma non è usato tanto spesso quanto la concatenazione.

L'operatore \* può essere usato solo con due valori numerici (per la moltiplicazione) o con un valore stringa e un intero (per la replica della stringa). In ogni altro caso, Python visualizzerà un messaggio d'errore.

```
>>> 'Alice' * 'Bob'  
Traceback (most recent call last):  
File "<pyshell#32>", line 1, in <module>  
'Alice' * 'Bob'  
TypeError: can't multiply sequence by non-int of type 'str'  
>>> 'Alice' * 5.0  
Traceback (most recent call last):  
File "<pyshell#33>", line 1, in <module>  
'Alice' * 5.0
```

TypeError: can't multiply sequence by non-int of type 'float'

Si può capire che Python non sappia che cosa fare con queste espressioni: non si possono moltiplicare due parole, ed è difficile replicare una stringa arbitraria un numero frazionario di volte.

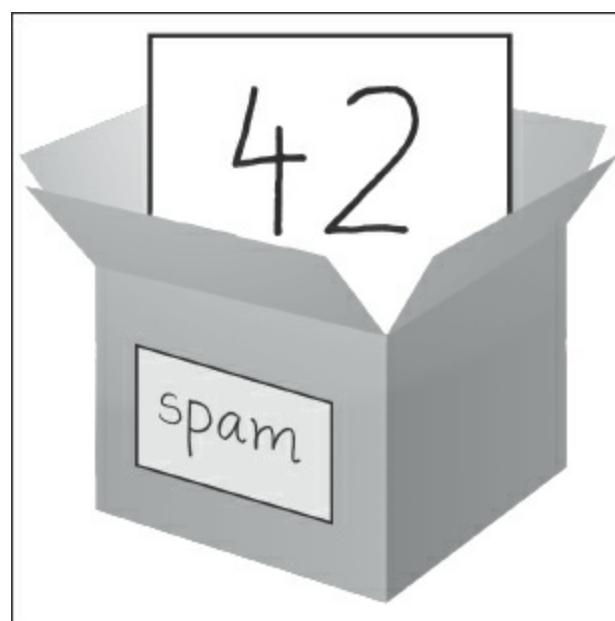
## Memorizzare valori in variabili

Una **variabile** è come un contenitore nella memoria del computer in cui si può depositare un singolo valore. Se più avanti nel programma si vuole usare un'espressione valutata, si può salvarla in una variabile.

## Enunciati di assegnazione

I valori si salvano nelle variabili mediante un **enunciato di assegnazione**, che è costituito dal nome di una variabile, da un segno di uguaglianza (chiamato **operatore di assegnazione**) e dal valore da memorizzare. Se inserite l'enunciato di assegnazione `spam = 42`, una variabile chiamata `spam` conterrà il valore intero 42.

Potete immaginare una variabile come una scatola con un'etichetta, all'interno della quale viene collocato un valore, come nella [Figura 1.2](#).



**Figura 1.2** - Scrivere `spam = 42` è come dire al programma “La variabile `spam` ora contiene il valore 42”.

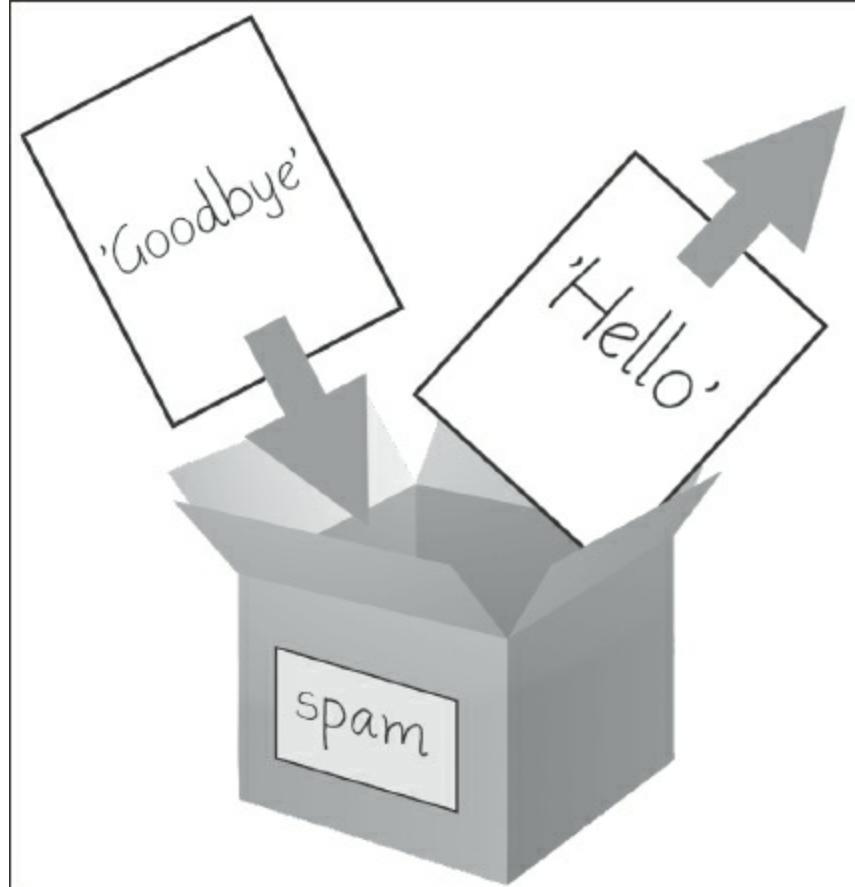
Per esempio, inserite nella shell interattiva quanto segue:

```
❶ >>> spam = 40
>>> spam
40
❷ >>> eggs = 2
>>> spam + eggs
42
>>> spam + eggs + spam
82
❸ >>> spam = spam + 2
>>> spam
42
```

Una variabile viene **inizializzata** (ovvero creata) la prima volta che vi viene memorizzato un valore **1**. Poi la si può usare in espressioni con altre variabili e valori **2**. Quando a una variabile viene assegnato un nuovo valore **3**, il vecchio valore viene dimenticato, ed è questo il motivo per cui `spam` viene valutata 42 anziché 40 alla fine dell'esempio. In questo caso si dice che si *sovrascrive* la variabile. Inserite il codice seguente nella shell interattiva, per cercare di sovrascrivere una stringa:

```
>>> spam = 'Hello'  
>>> spam  
'Hello'  
>>> spam = 'Goodbye'  
>>> spam  
'Goodbye'
```

Come la scatola della [Figura 1.3](#), la variabile `spam` in questo esempio contiene il valore 'Hello', finché non viene sostituito da 'Goodbye'.



**Figura 1.3** - Quando a una variabile viene assegnato un nuovo valore, il vecchio viene “dimenticato”.

La [Tabella 1.3](#) presenta esempi di **nomi di variabili leciti**. A una variabile si può attribuire qualsiasi nome, purché vengano rispettate le tre regole seguenti.

1. Deve essere costituito da una parola sola (niente spazi, quindi).
2. Può usare esclusivamente lettere, numeri e il carattere *underscore*, ovvero di sottolineatura (\_).
3. Non può iniziare con un numero.

**Tabella 1.3** - Nomi di variabile validi e non.

Nomi di variabile validi	Nomi di variabile non validi
saldo	saldo-corrente (i trattini non sono consentiti)
saldoCorrente	saldo corrente (gli spazi non sono consentiti)
saldo_corrente	4conto (non può iniziare con un numero)
_spam	42 (non può iniziare con un numero)
SPAM	totale_ \$omma (non sono consentiti caratteri speciali come \$)
Conto4	'ciao' (caratteri speciali come ' non sono consentiti)

I nomi delle variabili sono **case-sensitive**, cioè distinguono fra maiuscole e minuscole, il che significa che spam, SPAM, Spam e sPaM sono quattro variabili diverse. È una convenzione di Python che i nomi delle variabili inizino sempre con una lettera minuscola.

Questo libro usa le lettere maiuscole all'interno dei nomi composti di variabili (in inglese si usa l'espressione *camelcase*, ortografia "a cammello" per indicare questa pratica), anziché il trattino di sottolineatura, cioè variabiliComeQuesta, anziché variabili\_comme\_questa. Qualche programmatore esperto potrà obiettare che la guida ufficiale di stile per il codice Python, PEP 8, sostiene che si debbano usare i trattini di sottolineatura. Preferisco fare a modo mio e citare un paragrafo della stessa guida PEP 8:

La coerenza con la guida di stile è importante. Ma è ancora più importante sapere quando non esserne coerenti – a volte la guida di stile semplicemente non si applica. Quando siete in dubbio, usate la vostra testa.

Un buon nome di variabile descrive i dati che contiene. Immaginatevi di traslocare in una nuova casa e di mettere tutte le vostre cose in scatoloni etichettati tutti *Cose*. Non riuscireste più a trovare niente! Come nomi generici di variabili in questo libro, come in gran parte della documentazione di Python, vengono usati spam, eggs e bacon (l'ispirazione arriva da una scenetta dei Monty Python), ma nei vostri programmi un **nome descrittivo** contribuirà a rendere più leggibile il vostro codice.

## Il primo programma

La shell interattiva va benissimo per eseguire istruzioni Python una alla volta, ma per scrivere interi programmi Python dovete scrivere le istruzioni nel **file editor**. Il file editor è simile a editor di testo come Blocco note o TextMate, ma ha in più alcune caratteristiche specifiche per il codice sorgente. Per aprire il file editor in IDLE, selezionate **File > New File**.

La finestra che si aprirà conterrà un cursore che attende il vostro input, ma è diversa dalla shell interattiva, che esegue le istruzioni Python non appena premete Invio. Il file editor permette di scrivere molte istruzioni, salvare il file e quindi eseguire il programma. Ecco come potete distinguere i due tipi di finestra:

- la finestra della shell interattiva sarà sempre quella con il prompt **>>>**;
- la finestra del file editor non ha il prompt **>>>**.

Ora è il momento di creare il vostro primo programma! Quando si apre la finestra del file editor, scrivete al suo interno quanto segue:

```
print('Ciao mondo!')
print('Qual è il tuo nome?') # chiede il nome
mioNome = input()
print('È bello fare la tua conoscenza, ' + mioNome)
print('La lunghezza del tuo nome è: ')
print(len(mioNome))
print('Quanti anni hai?') # chiede l'età
miaEta = input()
print('Avrai ' + str(int(miaEta) + 1) + ' anni fra un anno.')
```

Una volta inserito il codice, salvatelo, così non dovete riscriverlo ogni volta che avviate IDLE. Dal menu nella parte superiore della finestra del file editor, selezionate **File > Save As**. Nella finestra *Salva con nome*, inserite *hello.py* nel campo *Nome file*, poi fate clic su **Salva**.

È bene salvare i programmi periodicamente, mentre li si scrive. In questo modo, se il computer si blocca o se uscite per sbaglio da IDLE, il codice non va perso. Come scocciatoia, potete premere Ctrl-S in Windows e Linux oppure ⌘-S in OS X per salvare il vostro file.

Una volta salvato, provate a eseguire il programma. Selezionate **Run > Run Module** oppure semplicemente premete il tasto F5. Il vostro programma verrà eseguito nella finestra della shell che si è aperta quando avete lanciato IDLE. Ricordate: dovete premere F5 dalla finestra del file editor, non dalla finestra della shell. Inserite il vostro nome quando il programma ve lo chiede. L'output del programma nella shell interattiva dovrebbe essere qualcosa di simile a questo:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC v.1900 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Ciao mondo!
Qual è il tuo nome?
A1
È bello fare la tua conoscenza, A1
La lunghezza del tuo nome è:
2
Quanti anni hai?
4
Avrai 5 anni fra un anno.
>>>
```

Quando non ci sono più righe di codice da eseguire, il programma Python *termina*: in altre parole, smette di girare, si conclude. (Si dice anche che il programma *esce*.)

Potete chiudere il file editor facendo clic sulla X in alto a destra nella finestra. Per ricaricare un programma salvato, selezionate dal menu **File > Open**. Provateci ora e, nella finestra che compare, scegliete *hello.py* e fate clic sul pulsante *Apri*. Il programma *hello.py* che avete salvato dovrebbe riaprirsi nella finestra del file editor.

## Analisi del programma

Con il vostro nuovo programma aperto nel file editor, diamo una rapida occhiata alle istruzioni Python che utilizza, esaminando una riga di codice alla volta.

# Commenti

La riga seguente è un **commento**.

```
❶ # Questo programma mi saluta e mi chiede il mio nome.
```

Python ignora i commenti, e potete utilizzarli per scrivere **annotazioni** o promemoria per ricordarvi che cosa fa il codice.

Tutto il testo che segue un segno di cancelletto o diesis (#), fino al termine della riga, fa parte del commento.

A volte i programmatore mettono un # davanti a una riga di codice per eliminarla temporaneamente mentre mettono alla prova un programma. Si parla allora di **trasformare in commento** (*commenting out*, in inglese) il codice, e può essere un buon metodo, quando si vuole capire perché un programma non funziona. Poi si può togliere il #, quando si è pronti a rimettere in funzione quella riga di codice. Python ignora anche la riga vuota dopo il commento. Potete aggiungere al vostro programma tutte le righe vuote che volete, per rendere il codice più facile da leggere.

## La funzione print()

La funzione `print()` visualizza sullo schermo il valore stringa racchiuso fra le parentesi.

```
❷ print('Ciao mondo!')
    print('Qual è il tuo nome?')  # chiede il nome
```

La riga `print('Ciao mondo!')` significa “Stampa il testo nella stringa 'Ciao mondo!'”. Quando Python esegue questa riga, si dice che **chiama la funzione** `print()` e che il valore stringa viene *passato* alla funzione. Un valore che viene passato a una chiamata di funzione è un **argomento**. Notate che gli apici non vengono visualizzati sullo schermo: servono solamente a indicare dove la stringa inizia e dove finisce, non fanno parte del valore.

### NOTA

Potete usare questa funzione anche per lasciare una riga vuota sullo schermo: basta chiamare `print()` senza inserire nulla fra le parentesi.

Quando si scrive il nome di una funzione, la coppia di parentesi, aperta e chiusa, alla fine lo identifica come nome di una funzione. Per questo scriveremo sempre `print()` anziché solo `print`. Il [Capitolo 2](#) descrive le funzioni più dettagliatamente.

## La funzione input()

La funzione `input()` attende che l’utente scriva qualcosa alla tastiera e poi prema Invio.

```
❸ mioNome = input()
```

Questa chiamata di funzione ha come valore una stringa uguale al testo inserito dall’utente e la riga di

codice assegna questo valore stringa alla variabile `mioNome`. Potete pensare la chiamata di `input()` come un'espressione che viene valutata alla stringa inserita dall'utente. Se l'utente scrive 'Al', il valore dell'espressione sarà `mioNome = 'Al'`.

## Stampare il nome dell'utente

La successiva chiamata di `print()` contiene tra le parentesi l'espressione '`È bello fare la tua conoscenza, ' + mioNome`'.

```
❸ print('È bello fare la tua conoscenza, ' + mioNome)
```

Ricordate che le espressioni possono sempre essere valutate a un valore singolo. Se 'Al' è il valore memorizzato in `mioNome` nella riga precedente, allora il valore di questa espressione sarà '`È bello fare la tua conoscenza, Al`'. Questo singolo valore stringa viene poi passato a `print()`, che lo stampa sullo schermo.

## La funzione `len()`

Alla funzione `len()` si può passare un valore stringa (o una variabile contenente una stringa) e la funzione dà il valore intero del numero dei caratteri che costituiscono quella stringa.

```
❹ print('La lunghezza del tuo nome è: ')
      print(len(mioNome))
```

Inserite nella shell quanto segue, per provare:

```
>>> len('hello')
5
>>> len('Il mio energico mostro ha mangiato nachos')
41
>>> len('')
0
```

Come questi esempi, `len(mioNome)` viene valutata a un intero. Poi questo viene passato a `print()` per essere visualizzato sullo schermo. Notate che a `print()` è consentito passare o valori interi o valori stringa. Notate però l'errore che viene visualizzato se si scrive questo nella shell:

```
>>> print('Ho ' + 29 + ' anni.')
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print('Ho ' + 29 + ' anni.')
TypeError: Can't convert 'int' object to str implicitly
```

L'errore non è causato dalla funzione `print()`, ma dall'espressione che avete tentato di passare a `print()`. Otterrete lo stesso messaggio d'errore se scrivete l'espressione da sola nella shell.

```
>>> 'Ho ' + 29 + ' anni.'
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    'Ho ' + 29 + ' anni.'
TypeError: Can't convert 'int' object to str implicitly
```

Python visualizza un messaggio d'errore perché si può usare l'operatore + solo per sommare fra loro due interi o per concatenare due stringhe. Non è possibile sommare un intero a una stringa, perché la grammatica di Python non lo consente. Potete risolvere il problema utilizzando una versione stringa dell'intero, come è spiegato nel prossimo paragrafo.

## Le funzioni str(), int() e float()

Se volete **concatenare un intero** come 29 a **una stringa**, per passare il risultato a `print()`, dovete ottenere il valore '29', che è la forma stringa di 29. Alla funzione `str()` si può passare un valore intero e darà come valore una versione stringa di quell'intero:

```
>>> str(29)
'29'
>>> print('Ho ' + str(29) + ' anni.')
Ho 29 anni.
```

Poiché `str(29)` viene valutata a '29', l'espressione `'Ho ' + str(29) + ' anni.'` viene valutata a `'Ho ' + '29' + ' anni.'`, che a sua volta viene valutata a `'Ho 29 anni.'`. Questo è il valore che viene passato alla funzione `print()`.

Le funzioni `str()`, `int()` e `float()` vengono valutate dando le forme stringa, intero e virgola mobile, rispettivamente, del valore che viene loro passato. Provate a convertire qualche valore nella shell interattiva con queste funzioni, e osservate che cosa accade.

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

Gli esempi precedenti chiamano le funzioni `str()`, `int()` e `float()` e passano loro valori degli altri tipi di dati per ottenere quegli stessi valori ma sotto forma rispettivamente di stringa, di intero e di numero in virgola mobile.

La funzione `str()` è comoda, quando si ha un numero intero o in virgola mobile che si vuole concatenare a una stringa. La funzione `int()` è utile se si ha un numero in forma di valore stringa e si vuole usarlo in qualche operazione matematica. Per esempio, la funzione `input()` restituisce sempre una stringa, anche se l'utente inserisce un numero. Inserite nella shell `spam = input()` e poi 101 quando resta in attesa del vostro `input`.

```
>>> spam = input()
101
```

```
>>> spam  
'101'
```

Il valore memorizzato in `spam` non è l'intero 101 ma la stringa '101'. Se volete eseguire qualche operazione matematica utilizzando il valore conservato in `spam`, usate la funzione `int()` per ottenere la forma intera di `spam` e poi memorizzatela come nuovo valore di `spam`.

```
>>> spam = int(spam)  
>>> spam  
101
```

Ora potrete trattare la variabile `spam` come un intero e non più come una stringa.

```
>>> spam * 10 / 5  
202.0
```

Notate che, se passate a `int()` un valore che non può valutare come un intero, Python presenta un messaggio d'errore.

```
>>> int('99.99')  
Traceback (most recent call last):  
  File "<pyshell#0>", line 1, in <module>  
    int('99.99')  
ValueError: invalid literal for int() with base 10: '99.99'  
>>> int('dodici')  
Traceback (most recent call last):  
  File "<pyshell#1>", line 1, in <module>  

```

La funzione `int()` è utile anche quando si vuole arrotondare un valore in virgola mobile.

```
>>> int(7.7)  
7  
>>> int(7.7) + 1  
8
```

Nel vostro programma, avete utilizzato le funzioni `int()` e `str()`, nelle ultime tre righe, per ottenere un valore del tipo di dati appropriato per il vostro codice.

```
❶ print('Quanti anni hai?')  # chiede l'età  
miaEta = input()  
print('Avrai ' + str(int(miaEta) + 1) + 'anni fra un anno.')
```

La variabile `miaEta` contiene il valore restituito da `input()`. Dato che la funzione `input()` restituisce sempre una stringa (anche se l'utente ha scritto un numero), potete usare `int(miaEta)` per trasformare in un valore intero la stringa contenuta in `miaEta`. Quel valore intero viene poi sommato a 1 nell'espressione `int(miaEta) + 1`.

Il risultato di questa addizione viene passato alla funzione `str()`: `str(int(miaEta) + 1)`. Il valore stringa restituito viene poi concatenato con le stringhe 'Avrai' e 'fra un anno.' e il valore finale sarà una lunga stringa, che infine viene passata a `print()` perché venga visualizzata sullo schermo.

Supponiamo che l'utente scriva la stringa '4' per miaEta. Quella stringa poi viene convertita in un intero, così che è possibile sommarvi 1. Il risultato è 5.

La funzione str() converte il risultato nuovamente in una stringa, in modo da poterlo concatenare con la seconda stringa, 'anni fra un anno.' e creare così il messaggio finale. Questi passi di valutazione avranno l'andamento visualizzato nella [Figura 1.4](#).

```
print('Avrai ' + str(int(miaEta) + 1) + ' anni fra un anno.')
print('Avrai ' + str(int( '4' ) + 1) + ' anni fra un anno.')
print('Avrai ' + str( 4 + 1 ) + ' anni fra un anno.')
print('Avrai ' + str( 5 ) + ' anni fra un anno.')
print('Avrai ' + '5' + ' anni fra un anno.')
print('Avrai 5' + ' anni fra un anno.')
print('Avrai 5 anni fra un anno.'
```

**Figura 1.4** - I passi di valutazione, nel caso in cui la variabile miaEta ha il valore 4.

### Equivaleanza fra testo e numeri

Anche se il valore stringa di un numero è considerato un valore completamente diverso dalla relativa versione intera o in virgola mobile, un intero può essere uguale a un valore in virgola mobile.

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

Python fa questa distinzione perché le stringhe sono testo, mentre sia i valori interi che quelli in virgola mobile sono entrambi numeri.

## Riepilogo

Potete **calcolare** espressioni con una calcolatrice o scrivere **concatenazioni di stringhe** con un word processor. È possibile anche **replicare** facilmente **stringhe** copiando e incollando testo. Ma le espressioni, e gli elementi che le compongono (operatori, variabili e chiamate di funzione) sono i "mattoni da costruzione" dei programmi. Una volta che saprete come maneggiare questi elementi, sarete in grado di dire a Python come agire per voi su grandi quantità di dati.

È importante ricordare i diversi tipi di **operatori** (+, -, \*, /, //, % e \*\*) per le operazioni matematiche, + e \* per le operazioni su stringhe) e i tre **tipi di dati** (interi, virgola mobile, stringhe) introdotti in questo capitolo.

Abbiamo introdotto anche alcune **funzioni**. Le funzioni print() e input() gestiscono semplice output testuale (sullo schermo) e input (dalla tastiera), rispettivamente. La funzione len() prende come

argomento una stringa e dà come valore un intero che è il numero dei caratteri contenuti nella stringa. Le funzioni `str()`, `int()` e `float()` restituiscono la forma stringa, intera o in virgola mobile, rispettivamente, dell'argomento che viene loro passato.

Nel prossimo capitolo, vedremo come dire a Python di prendere decisioni intelligenti sul codice da eseguire, su quello da saltare e quello da ripetere sulla base dei valori che ha a disposizione. È quello che si definisce *controllo del flusso* e permette di scrivere programmi che prendono decisioni intelligenti.

## Un po' di pratica

- Quali dei seguenti sono operatori, e quali sono valori?

\*

'hello'

-88.8

-

/

+

5

- Quale delle seguenti è una variabile e quale è una stringa?

spam

'spam'

- Indicate il nome di tre tipi di dati.
- Da che cosa è costituita un'espressione? Che cosa fanno tutte le espressioni?
- Questo capitolo ha presentato gli enunciati di assegnazione, come `spam = 10`. Qual è la differenza fra un'espressione e un enunciato?
- Che cosa contiene la variabile `bacon` dopo che è terminata l'esecuzione di questo codice?

`bacon = 20`

`bacon + 1`

- Qual è il valore delle due espressioni seguenti?

`'spam' + 'spamspam'`

`'spam' * 3`

- Perché `eggs` è un nome di variabile valido, mentre `100` non lo è?
- Quali sono tre funzioni che possono essere utilizzate per ottenere il valore intero, in virgola mobile o stringa di un valore dato?
- Perché l'espressione seguente provoca un errore? Come si può correggerla?

`'Ho mangiato ' + 99 + ' panzerotti.'`

**Credito extra:** Cercate online la documentazione per la funzione `len()` di Python. La troverete in una pagina web intitolata “Built-in Functions”. Scorrete l'elenco delle altre funzioni di Python, cercate la funzione `round()`, esaminate che cosa fa e sperimentatela nella shell interattiva.

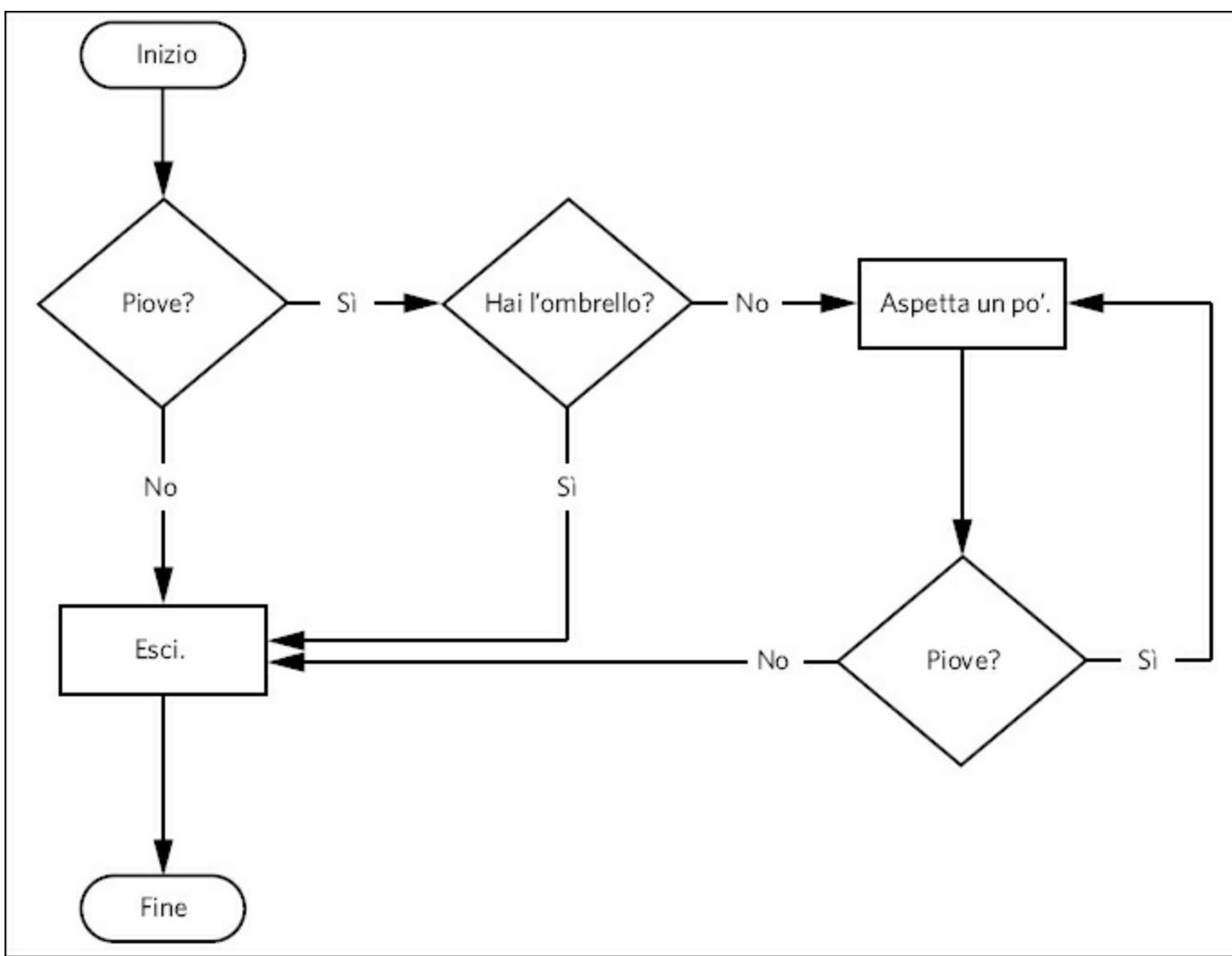
# Controllo del flusso

Ora conoscete gli aspetti fondamentali delle singole **istruzioni** e sapete che un **programma** è solo una serie di istruzioni. La vera potenza della programmazione però non sta nella semplice esecuzione di un'istruzione dopo l'altra, come se si trattasse della lista delle commissioni da sbrigare nel fine settimana. Sulla base del **valore delle espressioni**, il programma può **decidere** di saltare istruzioni, ripeterle, oppure sceglierne una fra varie possibili.

In effetti, quasi mai vorrete che i vostri programmi partano dalla prima riga di codice ed eseguano semplicemente ogni singola riga, una dopo l'altra, fino alla fine. **Enunciati di controllo del flusso** possono decidere quali istruzioni Python debba eseguire, a seconda delle condizioni.

Questi enunciati di controllo del flusso corrispondono direttamente ai simboli in un diagramma di flusso, perciò vi presenterò diagrammi di flusso del codice che analizzeremo in questo capitolo.

La [Figura 2.1](#) presenta un diagramma di flusso per quel che si deve fare quando piove. Seguite il percorso indicato dalle frecce, da *Inizio* a *Fine*.



**Figura 2.1** - Un diagramma di flusso per decidere che cosa fare se piove.

In un diagramma di flusso, di solito esistono più modi per andare dall'inizio alla fine, e lo stesso vale per le righe di codice di un programma per computer. I diagrammi di flusso rappresentano i **punti di diramazione** (dove si prende una **decisione**) con rombi, mentre gli altri passi sono rappresentati da rettangoli. I passi di inizio e fine sono rappresentati da rettangoli arrotondati.

Prima di poter parlare di enunciati di controllo del flusso, dovete prima imparare come rappresentare quelle opzioni *sì* e *no*, e dovete sapere come scrivere quei punti di diramazione sotto forma di codice Python. Per questo, cominciamo con l'esplorare i valori Booleani, gli operatori di confronto e gli operatori Booleani.

## Valori Booleani

Mentre i tipi di dati intero, in virgola mobile e stringa hanno un numero illimitato di valori possibili, il tipo di dato **Booleano** ha due soli valori possibili: vero e falso, ovvero, in Python, True e False. (Il termine “Booleano” viene scritto solitamente con l'iniziale maiuscola in omaggio al matematico da cui prende il nome, George Boole.) Quando vengono scritti come codice Python, i valori Booleani True e False non sono scritti fra apici come si fa per le stringhe, e hanno sempre l'iniziale maiuscola, mentre il resto della parola è in minuscolo. Inserite nella shell interattiva quel che segue. (Alcune di queste istruzioni sono intenzionalmente sbagliate, e provocheranno la comparsa di messaggi d'errore.)

```

❶ >>> spam = True
❷ >>> spam
True
❸ >>> true
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
true
NameError: name 'true' is not defined
❹ >>> True = 2 + 2
SyntaxError: assignment to keyword

```

Come qualsiasi altro valore, i Booleani vengono utilizzati nelle espressioni e possono essere memorizzati in variabili ❶. Se non utilizzate la giusta combinazione di maiuscole e minuscole ❷ o tentate di usare True o False come nomi di variabili ❸, Python vi comunica un errore.

## Operatori di confronto

Gli **operatori di confronto** confrontano due valori e danno come risultato un singolo valore Booleano.

Questi operatori sono elencati nella [Tabella 2.1](#).

**Tabella 2.1** - Operatori di confronto.

Operatore	Significato
<code>==</code>	Uguale a
<code>!=</code>	Non uguale a (diverso da)
<code>&lt;</code>	Minore di
<code>&gt;</code>	Maggiore di
<code>&lt;=</code>	Minore o uguale a
<code>&gt;=</code>	Maggiore o uguale a

Questi operatori vengono valutati True o False a seconda dei valori a cui vengono applicati. Facciamo qualche esperimento, partendo con `=` e `!=`.

```

>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False

```

Come si poteva prevedere, `=` (uguale a) viene valutato True quando i valori a cui è applicato sono uguali, e `!=` (non uguale a) viene valutato True quando i due valori sono diversi. Questi operatori possono essere applicati a valori di **qualsiasi tipo di dati**.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'cane' != 'gatto'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
❶ >>> 42 == '42'
False
```

Notate che un valore intero o in virgola mobile sarà sempre diverso da un valore stringa. L'espressione `42 == '42'` ❶ risulta `False` perché Python considera l'intero 42 diverso dalla stringa '42'. Gli operatori `<`, `>`, `<=` e `>=`, invece, funzionano solo con valori interi e in virgola mobile.

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
❶ >>> numeroUova = 42
>>> numeroUova <= 42
True
❷ >>> miaEta = 29
>>> miaEta >= 10
True
```

## La differenza fra gli operatori `==` e `=`

Avrete notato che l'operatore `=` (uguale a) è formato da due segni di uguaglianza, mentre l'operatore `=` (assegnazione) è costituito da un solo segno di uguaglianza. Non bisogna confondere fra loro questi due operatori. Ricordate sempre:

- L'operatore `=` (uguale a) chiede se due valori sono uguali fra loro.
- L'operatore `=` (assegnazione) attribuisce alla variabile che sta alla sua sinistra il valore che sta alla sua destra.

Per aiutarvi a distinguerli, notate anche che l'operatore `=` (uguale a) è formato da due caratteri, proprio come l'operatore `!=` (non uguale a).

Vi capiterà spesso di usare gli operatori di confronto per confrontare il valore di una variabile con qualche altro valore, come negli esempi `numeroUova <= 42` ❶ e `miaEta >= 10` ❷. (In fin dei conti, invece di scrivere `'cane' != 'gatto'`, avreste potuto scrivere direttamente `True` nel vostro codice.) Vedrete altri esempi quando parleremo degli enunciati di controllo del flusso.

## Operatori Booleani

I tre operatori Booleani (`and`, `or` e `not`) si usano per **confrontare valori Booleani**. Come gli operatori di

confronto, valutano le espressioni e danno un singolo risultato, che è un valore Booleano. Vediamoli in dettaglio, a partire dall'operatore `and`.

## Operatori Booleani binari

Gli operatori `and` e `or` si applicano sempre a due valori Booleani (o espressioni Booleane), perciò sono definiti **operatori binari**. L'operatore `and` valuta True un'espressione se entrambi i valori Booleani sono True; altrimenti la valuta False.

Provate a inserire qualche espressione utilizzando l'operatore `and` nella shell interattiva, per vederlo in azione.

```
>>> True and True
```

True

```
>>> True and False
```

False

Una *tavola di verità* mostra tutti i possibili risultati di un operatore Booleano. La [Tabella 2.2](#) presenta la tavola di verità per l'operatore `and`.

**Tabella 2.2** - La tavola di verità dell'operatore `and`.

Espressione	Valore
True and True	True
True and False	False
False and True	False
False and False	False

L'operatore `or` invece valuta True un'espressione se almeno uno dei due valori Booleani è True. Dà come valore False solo se entrambe le espressioni sono False.

```
>>> False or True
```

True

```
>>> False or False
```

False

Potete vedere tutti i possibili risultati dell'operatore `or` nella sua tavola di verità, presentata nella [Tabella 2.3](#).

**Tabella 2.3** - La tavola di verità dell'operatore `or`.

Espressione	Valore
True and True	True
True and False	True
False and True	True
False and False	False

## L'operatore not

A differenza di `and` e `or`, l'operatore `not` si applica a un unico valore Booleano (o a un'unica espressione) e dà come risultato il valore opposto a quello del Booleano a cui è applicato.

```
>>> not True  
False  
❶ >>> not not not not True  
True
```

Un po' come si usa la doppia negazione parlando e scrivendo, si possono annidare più operatori `not` ❶, anche se non c'è mai nessuna ragione di farlo nei programmi reali. La [Tabella 2.4](#) presenta la tavola di verità per `not`.

**Tabella 2.4** - La tavola di verità dell'operatore `not`.

Espressione	Valore
<code>not True</code>	<code>False</code>
<code>not False</code>	<code>True</code>

## Operatori Booleani e di confronto usati insieme

Poiché gli operatori di confronto danno come risultato valori Booleani, è possibile utilizzarli in espressioni con gli operatori Booleani.

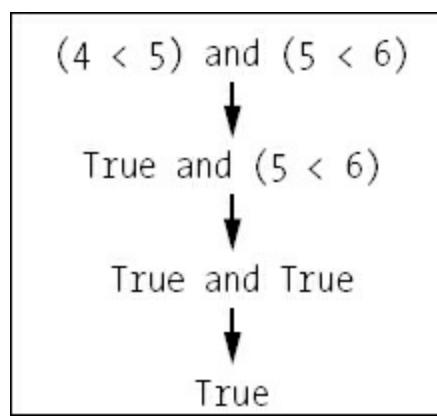
Come ricorderete, gli operatori `and`, `or` e `not` sono chiamati Booleani perché agiscono sempre sui valori Booleani `True` e `False`. Espressioni come `4 < 5` non sono valori Booleani, ma sono espressioni che, quando vengono valutate, danno come risultato valori Booleani.

Provate a inserire nella shell interattiva qualche espressione Booleana che usi operatori di confronto.

```
>>> (4 < 5) and (5 < 6)  
True  
>>> (4 < 5) and (9 < 6)  
False  
>>> (1 == 2) or (2 == 2)  
True
```

Il computer valuta prima l'espressione a sinistra, poi quella a destra. Quando conosce il valore Booleano di ciascuna, valuta l'espressione completa.

Potete pensare il processo di valutazione eseguito dal computer per `(4 < 5) and (5 < 6)` come è rappresentato nella [Figura 2.2](#).



**Figura 2.2** - Il processo di valutazione di una espressione contenente operatori di confronto e Booleani.

Potete usare in una stessa espressione anche più operatori Booleani e più operatori di confronto.

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

Anche per gli operatori Booleani, come per quelli matematici, esiste un **ordine di precedenza**: dopo che sono stati valutati tutti gli operatori matematici e quelli di confronto, Python valuta prima gli operatori `not`, poi gli operatori `and` e infine gli `or`.

## Elementi del controllo del flusso

Gli enunciati di controllo del flusso spesso iniziano con una parte che è chiamata *condizione*, e tutti sono seguiti da un blocco di codice, chiamato *clausola*. Prima di vedere gli specifici enunciati di controllo del flusso di Python, vediamo che cosa sono una condizione e un blocco.

## Condizioni

Le espressioni Booleane viste fin qui possono essere considerate tutte **condizioni**, che sono la stessa cosa delle espressioni: “condizione” è semplicemente un nome più specifico, nel contesto degli enunciati di controllo del flusso. Le condizioni **hanno sempre un valore Booleano**, True o False.

Un enunciato di controllo del flusso decide che cosa fare in base al fatto che la sua condizione sia True o False, e quasi tutti gli enunciati di controllo del flusso usano una condizione.

## Blocchi di codice

Le righe di codice Python possono essere raggruppate in **blocchi**. Si può stabilire dove inizia e finisce un blocco da come sono rientrate (o, come si dice anche, “indentate”) le righe di codice. Esistono tre **regole per i blocchi**.

1. I blocchi iniziano quando il rientro aumenta.
2. I blocchi possono contenere altri blocchi.
3. Un blocco finisce quando il rientro diminuisce a zero o si riduce all’indentazione del blocco che lo contiene.

È più facile capire che cos’è un blocco esaminando del codice con rientri.

Vediamo dunque di identificare i blocchi in questo frammento di un piccolo programma per un gioco:

```

❶ if name == 'Mary':
    print('Hello Mary')
    if password == 'swordfish':
❷     print('Accesso consentito.')
    else:
❸         print('Password errata.')

```

Il primo blocco di codice ❶ inizia con la riga `print('Hello Mary')` e contiene tutte le righe di codice che seguono.

All'interno di questo blocco vi è un altro blocco ❷, che è formato da una sola riga: `print('Accesso consentito.')`.

Anche il terzo blocco ❸ è formato da una sola riga: `print('Password errata.')`.

## Esecuzione del programma

Nel programma `hello.py` del capitolo precedente, Python iniziava a eseguire le istruzioni a partire dalla prima riga del codice e continuava in ordine, una riga dopo l'altra. **Esecuzione del programma** (o semplicemente esecuzione) è un termine che indica l'istruzione eseguita in quel momento. Se stampate il codice sorgente su carta e mettete il dito su ciascuna riga quando viene eseguita, potete pensare che il vostro dito sia l'esecuzione del programma.

Non tutti i programmi vengono eseguiti semplicemente procedendo in ordine dalla prima riga di codice all'ultima. Se usate il dito per seguire un programma con enunciati di controllo del flusso, con tutta probabilità vi troverete a saltare da un punto all'altro del codice sorgente sulla base delle condizioni, e probabilmente vi saranno casi in cui salterete intere clausole.

## Enunciati di controllo del flusso

Esploriamo ora l'elemento più importante del controllo del flusso: gli enunciati stessi. Questi enunciati corrispondono ai rombi visti nel diagramma di flusso della [Figura 2.1](#), e sono le decisioni effettive prese dai vostri programmi.

### Enunciati if

Il tipo più comune di enunciato di controllo del flusso è l'enunciato `if`. La clausola dell'enunciato `if` (cioè il blocco che segue l'enunciato `if`) verrà eseguita se la condizione nell'enunciato è `True`. Se la condizione è `False` la clausola viene saltata.

L'enunciato `if` può essere letto, in italiano, come “Se questa condizione è vera, esegui il codice contenuto nella clausola”. In Python, un enunciato `if` è costituito da:

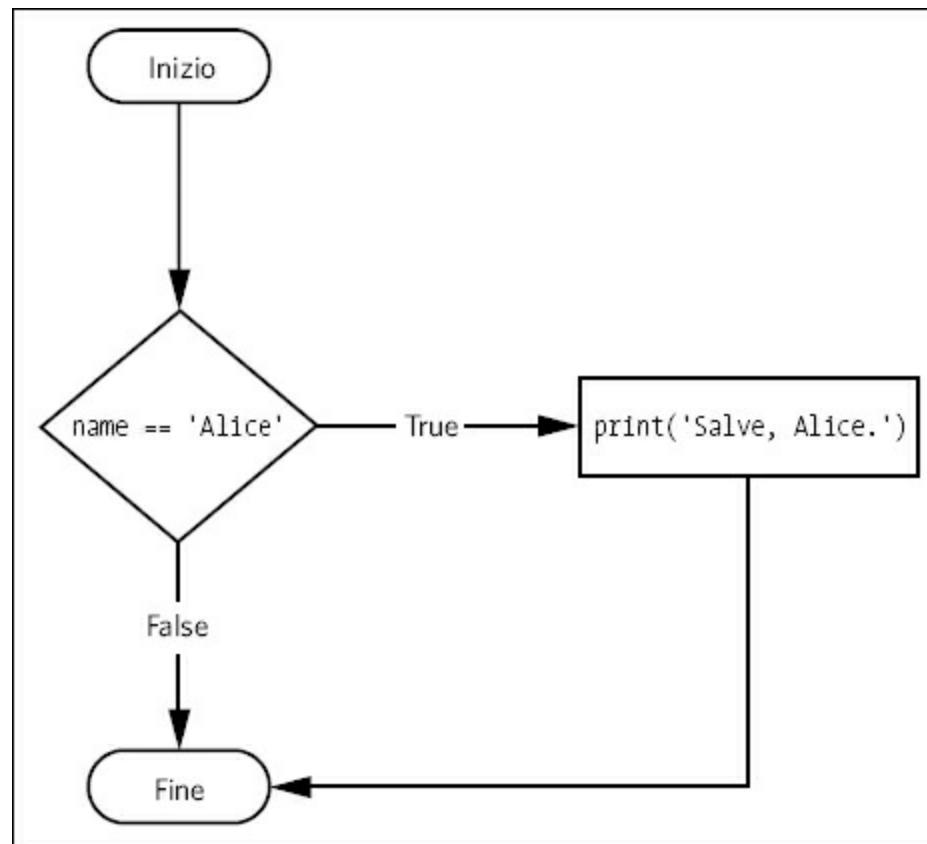
- la parola chiave `if`;
- una condizione (cioè un'espressione che ha valore `True` o `False`);
- un segno di due punti (`:`);
- a partire dalla riga successiva, un blocco di codice rientrato (la clausola `if`).

Per esempio, supponiamo di avere del codice che verifica se il nome di una persona è `Alice`. (Immaginiamo che alla variabile `nome` sia stato assegnato già in precedenza qualche valore.)

```
if nome == 'Alice':
```

```
print('Salve, Alice.')
```

Tutti gli enunciati di controllo del flusso terminano con un segno di due punti e sono seguiti da un nuovo blocco di codice (la clausola). La clausola dell'enunciato if è, nell'esempio, il blocco costituito dalla riga `print('Salve, Alice.')`. La [Figura 2.3](#) presenta il diagramma di flusso di questo piccolo spezzone di codice.



**Figura 2.3** - Il diagramma di flusso per un enunciato if.

## Enunciati else

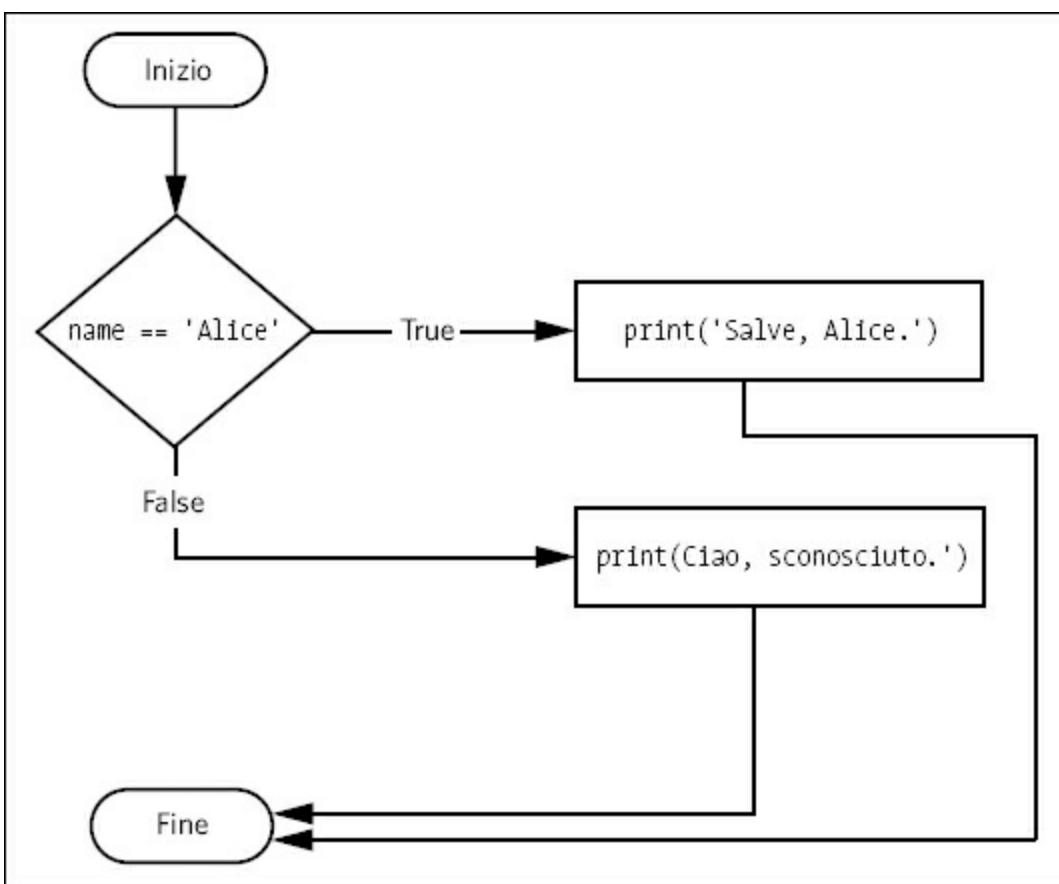
Facoltativamente, una clausola if può essere seguita da un enunciato else (che in inglese equivale all'italiano “altrimenti”). La clausola else viene eseguita solo quando la condizione nell'enunciato if è False. Un enunciato else equivale a dire: “Se questa condizione è vera, esegui questo codice. Altrimenti, esegui quest’altro codice”. Un enunciato else non ha una condizione e, nel codice, è costituito sempre da:

- la parola chiave `else`;
- un segno di due punti;
- a partire dalla riga successiva, un blocco di codice rientrato (la clausola `else`).

Per tornare all'esempio di Alice, vediamo un frammento di codice che utilizza un enunciato else per presentare un saluto diverso, se il nome della persona non è Alice.

```
if nome == 'Alice':  
    print ('Salve, Alice.')  
else:  
    print('Ciao, sconosciuto.')
```

La [Figura 2.4](#) mostra il diagramma di flusso per questo frammento di codice.



**Figura 2.4** - Il diagramma di flusso per un enunciato else.

## Enunciati elif

Nel caso di if ed else la scelta è fra l'esecuzione di una o dell'altra fra due clausole, ma ci sono situazioni in cui si può volere che la scelta sia fra molte clausole possibili. L'enunciato elif è un enunciato “else if” (ovvero “altrimenti se”) che segue sempre un if o un altro elif. Presenta un'altra condizione, che viene verificata solo se tutte le condizioni precedenti sono risultate False. Nel codice, un enunciato elif è sempre costituito da:

- la parola chiave elif;
- una condizione (cioè un'espressione che può avere valore True o False);
- un segno di due punti;
- a partire dalla riga successiva, un blocco di codice rientrato (la clausola elif).

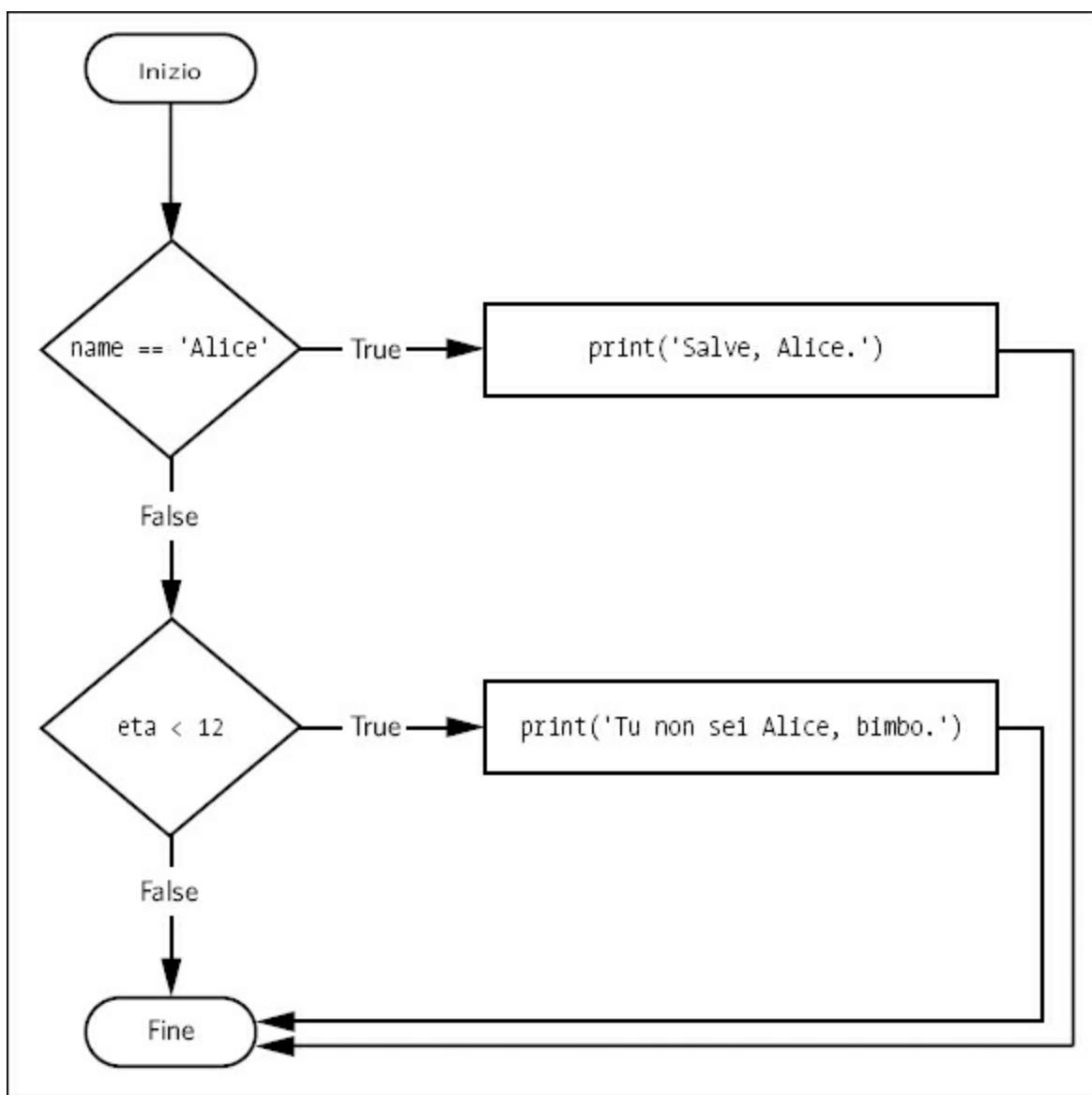
Aggiungiamo un elif al programma che verifica il nome, per vedere in azione questo tipo di enunciato:

```

if nome == 'Alice':
    print('Salve, Alice.')
elif eta < 12:
    print('Tu non sei Alice, bimbo.')

```

Questa volta, viene controllata l'età della persona, e il programma dà una risposta diversa, se la persona ha meno di 12 anni. Il diagramma di flusso è nella [Figura 2.5](#).



**Figura 2.5** - Il diagramma di flusso per un enunciato `elif`.

La clausola `elif` viene eseguita se `eta < 12` è `True` e `nome == 'Alice'` è `False`. Tuttavia, se entrambe le conclusioni sono `False`, entrambe le clausole vengono saltate. Non è detto che almeno una delle clausole venga eseguita. Quando vi è una serie di enunciati `elif`, solo una delle clausole viene eseguita, oppure nessuna. Non appena una delle condizioni degli enunciati viene trovata `True`, il resto delle clausole `elif` viene automaticamente saltato.

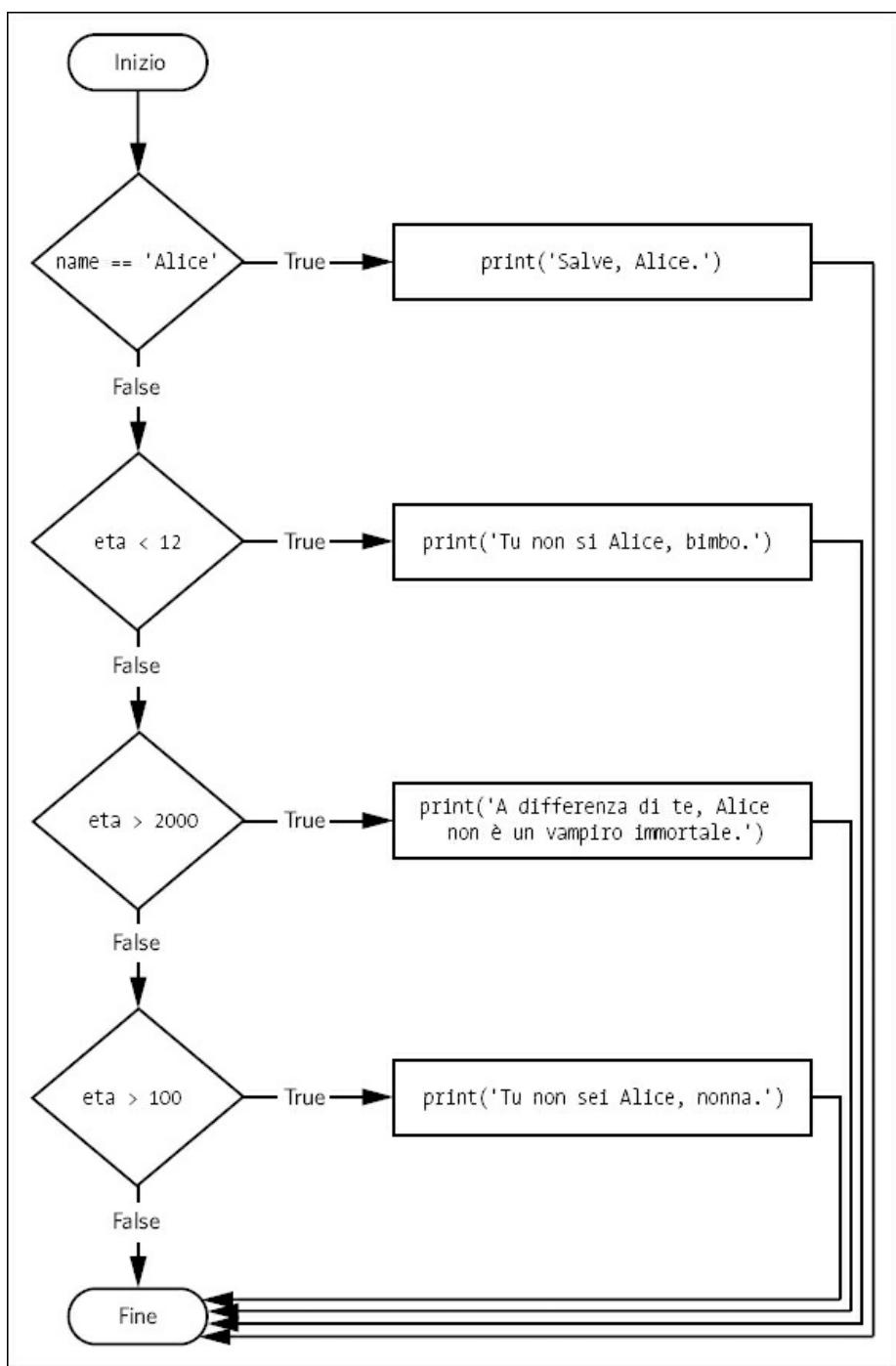
Per esempio, apriete una nuova finestra di file editor e inserite il codice seguente, poi salvatelo con il nome `vampire.py`:

```

if nome == 'Alice':
    print('Salve, Alice.')
elif age < 12:
    print('Tu non sei Alice, bimbo.')
elif eta > 2000:
    print('A differenza di te, Alice non è un vampiro immortale.')
elif eta > 100:
    print('Tu non sei Alice, nonna.')

```

Qui ho aggiunto altri due enunciati `elif` perché il programma di verifica del nome saluti una persona in modo diverso a seconda dell'età. La [Figura 2.6](#) mostra il diagramma di flusso relativo.



**Figura 2.6** - Il diagramma di flusso per i molti enunciati `elif` nel programma `vampire.py`.

L'ordine degli enunciati `elif` è importante. Proviamo a riorganizzarli, per introdurre un errore. Ricordate che tutte le successive clausole `elif` vengono automaticamente saltate, non appena viene trovata una condizione `True`, perciò se invertite l'ordine di alcune delle clausole di `vampire.py`, vi ritrovate con un problema.

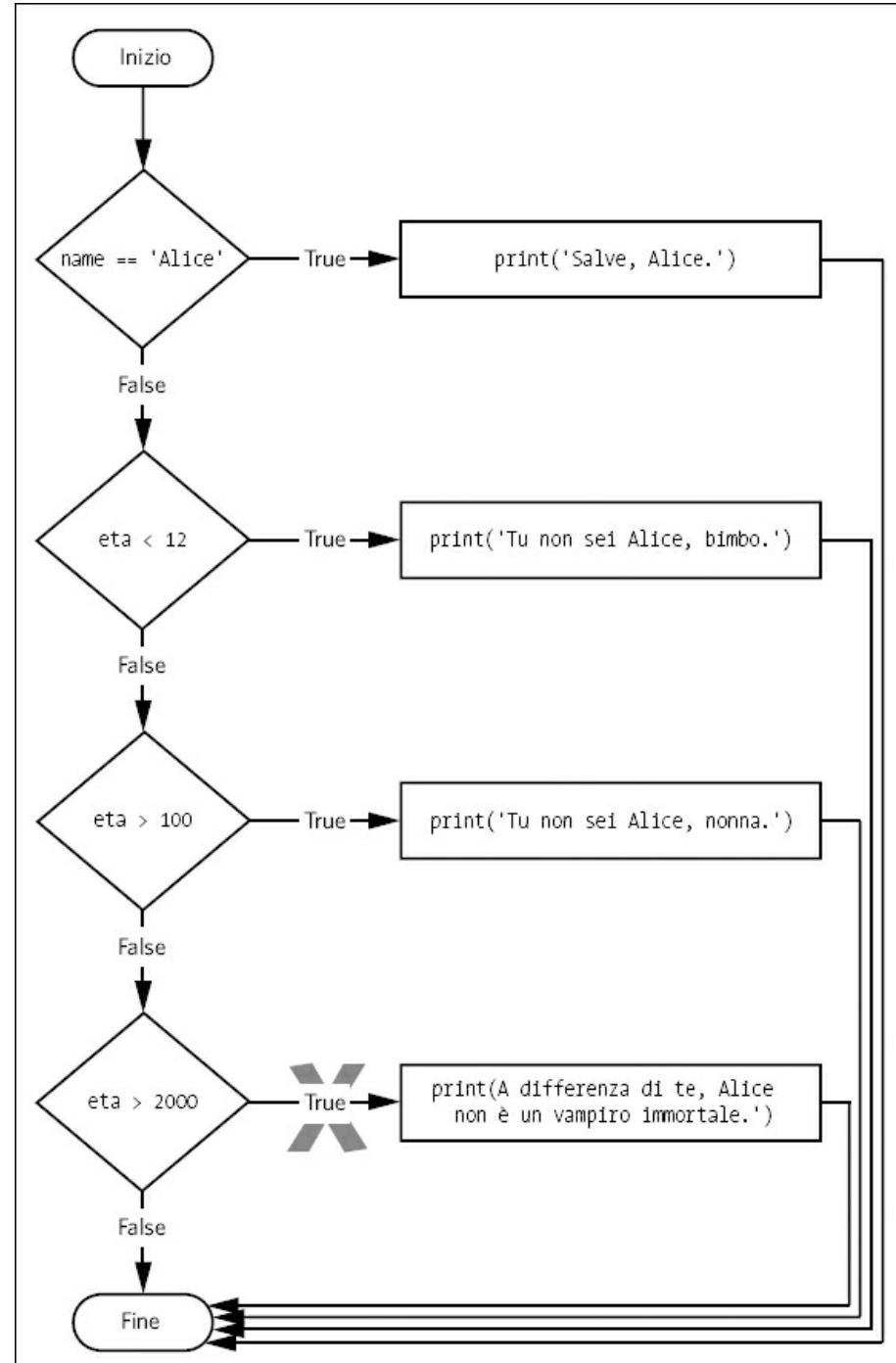
Modificate il codice in questo modo, poi salvate il file con il nuovo nome di `vampire2.py`:

```

if nome == 'Alice':
    print('Salve, Alice.')
elif age < 12:
    print('Non prendermi in giro, tu non sei Alice.')
❶ elif eta > 100:
    print('Tu non sei Alice, nonna.')
elif eta > 2000:
    print('A differenza di te, Alice non è un vampiro immortale.')
    
```

Supponiamo che, prima che venga eseguito questo pezzo di codice, la variabile `eta` contenga il valore 3000. Vi aspettereste che il codice stampi la stringa 'A differenza di te, Alice non è un vampiro immortale.', ma poiché la condizione `eta > 100` è True (in fin dei conti, 3000 è maggiore di 100) ❶, viene stampata la stringa 'Tu non sei Alice, nonna.' e il resto degli enunciati `elif` viene automaticamente saltato. Ricordate: verrà eseguita al più una delle clausole `e`, per gli enunciati `elif`, l'ordine è importante.

La Figura 2.7 presenta il diagramma di flusso per quest'ultimo frammento di codice. Notate che i rombi per `eta > 100` e `eta > 2000` sono stati scambiati di posto, rispetto alla Figura 2.6.



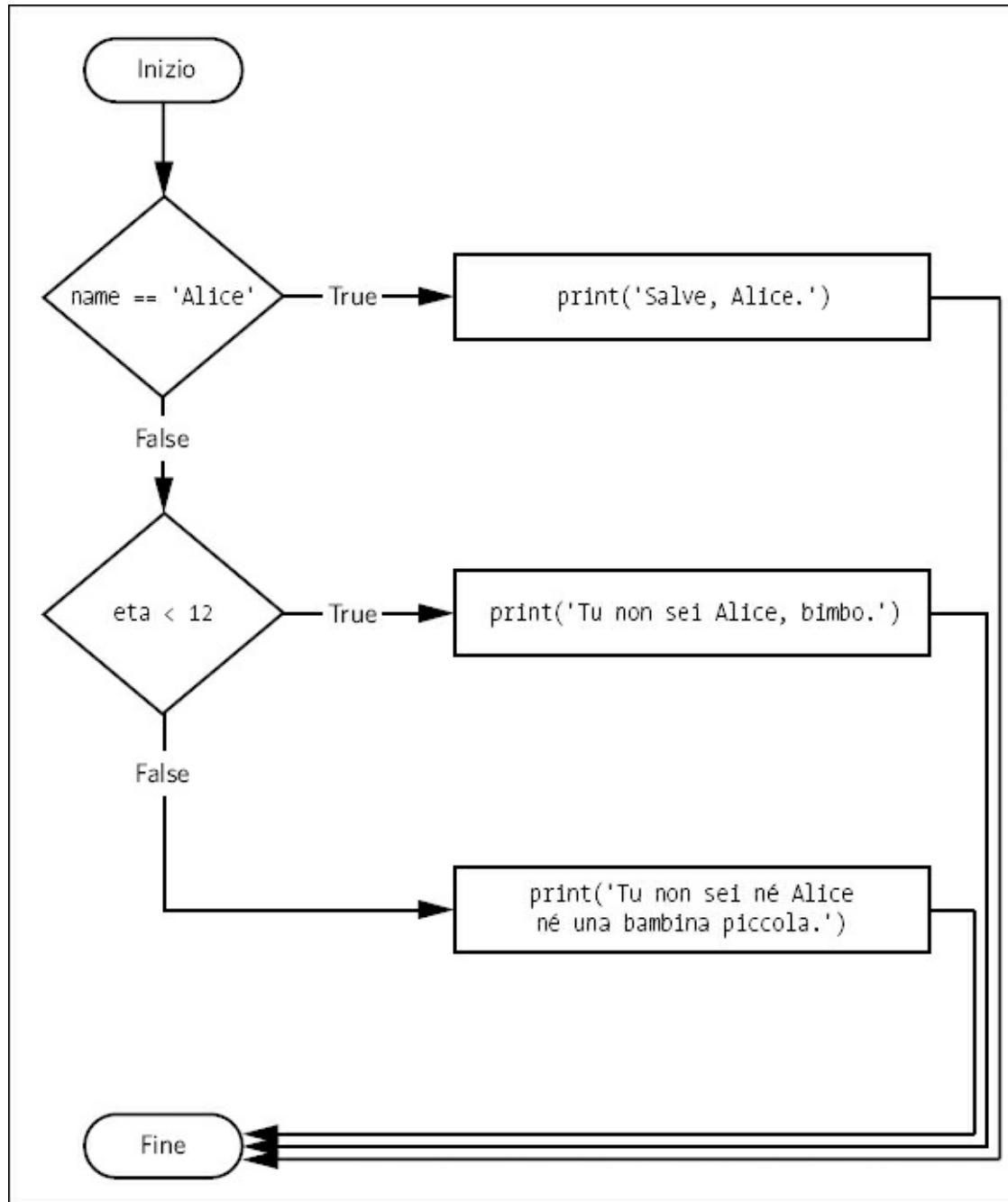
**Figura 2.7** - Il diagramma di flusso per il programma `vampire2.py`. Il percorso evidenziato dalla grande X non verrà mai imboccato, perché, se la variabile `eta` avesse un valore maggiore di 2000, risulterebbe già maggiore di 100.

Volendo, si può inserire un enunciato `else` dopo l'ultimo enunciato `elif`. In tal caso, è garantito che almeno una (e solo una) delle clausole verrà eseguita. Se le condizioni in tutti gli enunciati `if` ed `elif` risultano False, allora viene eseguita la clausola `else`.

Proviamo, per esempio, a ricreare il programma di Alice utilizzando clausole if, elif ed else.

```
if nome == 'Alice':  
    print('Salve, Alice.')  
elif eta < 12:  
    print('Tu non sei Alice, bimbo')  
else:  
    print('Tu non sei né Alice né una bambina piccola.')
```

La [Figura 2.8](#) presenta il diagramma di flusso per questo nuovo codice, che possiamo salvare con il nome *littleKid.py*.



**Figura 2.8** - Il diagramma di flusso per il programma *littleKid.py*.

In linguaggio di tutti i giorni, questo tipo di struttura di controllo del flusso significherebbe: “Se la prima condizione è vera, fai questo. Altrimenti, se è vera la seconda condizione, fai quest’altro. Altrimenti, se anche la seconda non è vera, fai quest’altra cosa ancora”. Se usate questi tre enunciati

insieme, ricordate sempre queste regole sul modo di ordinarli, per evitare errori come quello nella [Figura 2.7](#): in primo luogo, c'è sempre esattamente un solo enunciato if. Tutti gli enunciati elif che vi servono devono seguire l'enunciato if. In secondo luogo, se volete essere sicuri che venga eseguita almeno una clausola, chiudete la struttura con un enunciato else.

## Enunciati di ciclo while

Potete fare in modo che un blocco di codice venga eseguito più volte, utilizzando un enunciato while (che significa “mentre”). Il codice in una clausola while verrà eseguito continuamente per tutto il tempo per cui la condizione dell'enunciato while è True. Nel codice, un enunciato while è sempre costituito da:

- la parola chiave while;
- una condizione (cioè un'espressione che ha valore True o False);
- un segno di due punti;
- a partire dalla riga successiva, un blocco di codice rientrato (la clausola while).

Come potete vedere, un enunciato while assomiglia molto a un enunciato if. La differenza sta nel loro comportamento.

Al termine di una clausola if, l'esecuzione del programma continua dopo l'enunciato if; alla fine di una clausola while, invece, l'esecuzione del programma salta indietro di nuovo all'inizio dell'enunciato while.

La clausola while spesso viene chiamata **ciclo while** o semplicemente *ciclo* (*loop* in inglese).

Esaminiamo un enunciato if e un ciclo while che usano la stessa condizione e intraprendono le stesse azioni sulla base di quella condizione. Ecco il codice con un enunciato if:

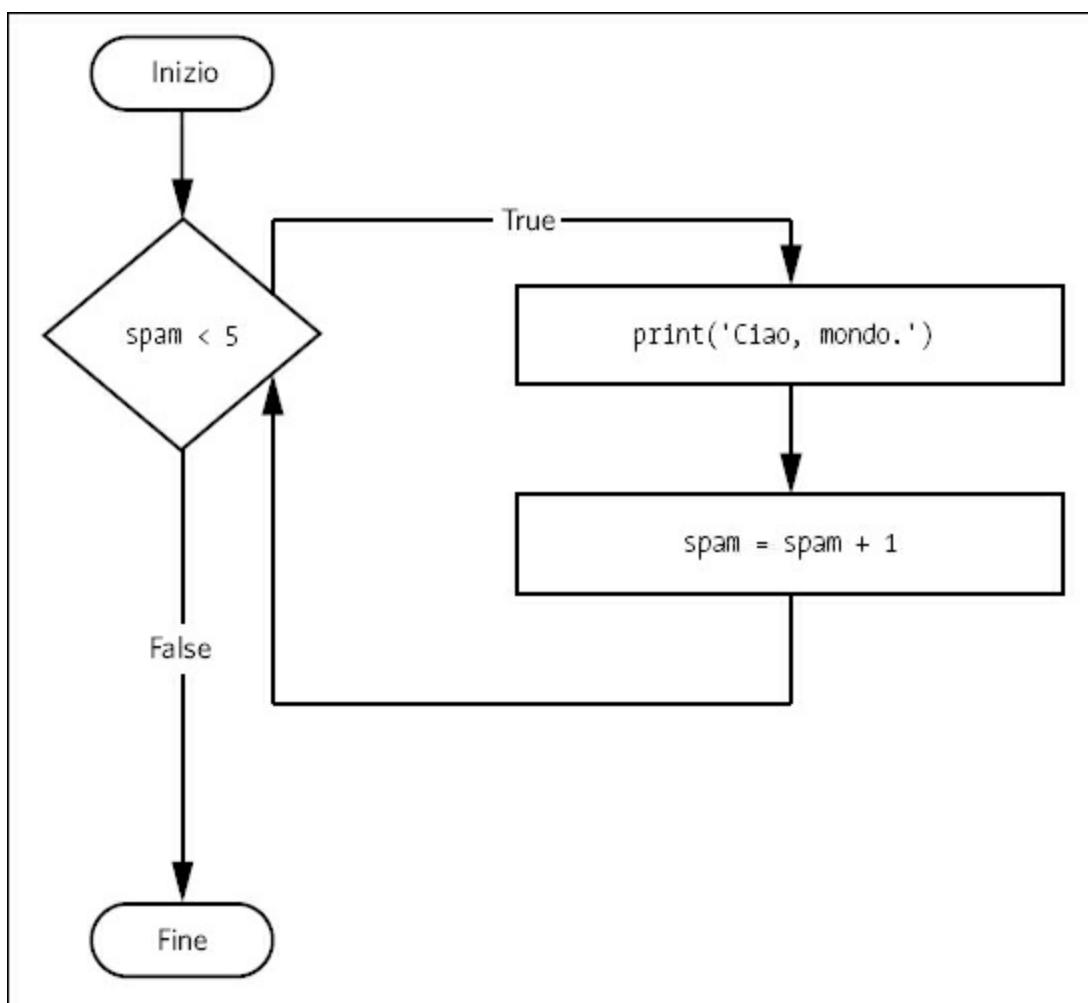
```
spam = 0
if spam < 5:
    print('Ciao, mondo.')
    spam = spam + 1
```

Ed ecco il codice con un enunciato while:

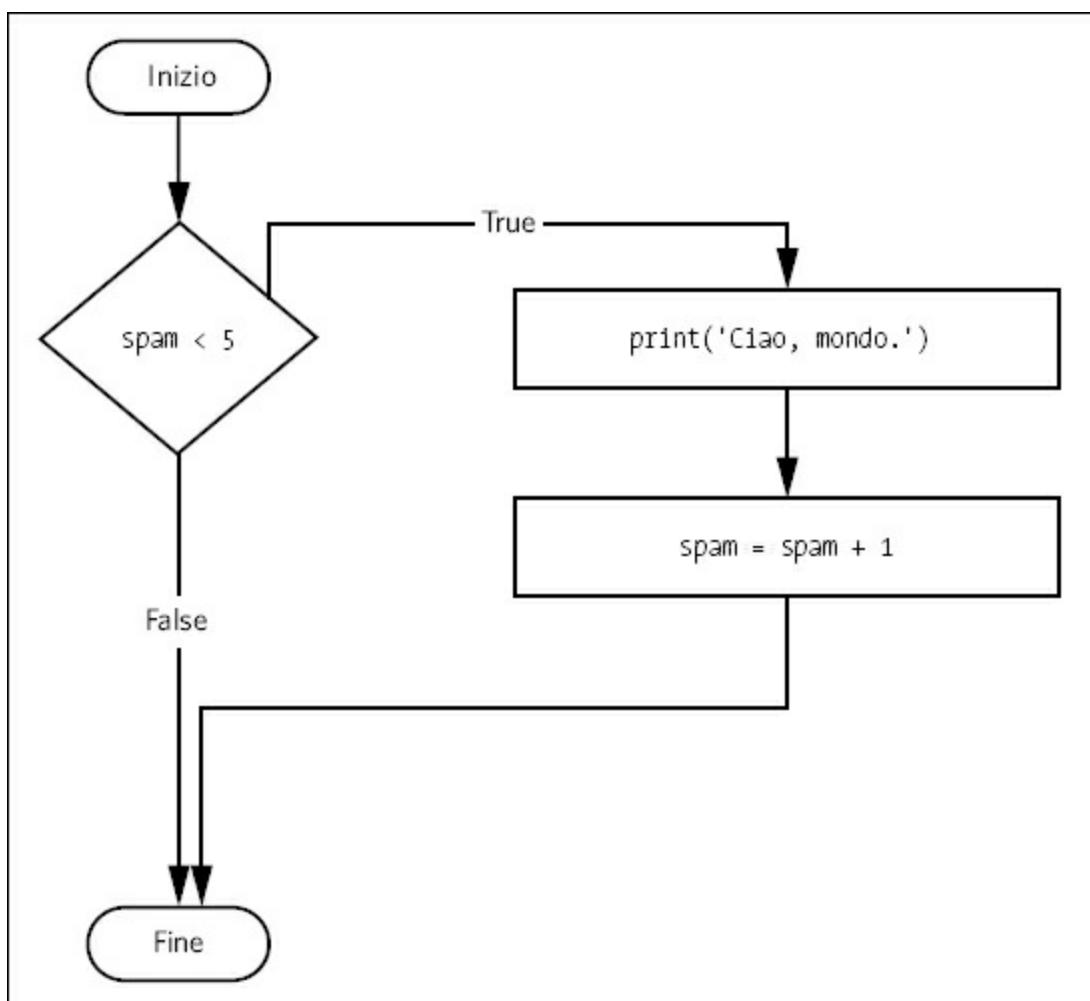
```
spam = 0
while spam < 5:
    print('Ciao, mondo.')
    spam = spam + 1
```

Questi enunciati sono simili: sia if che while verificano il valore di spam e, se è minore di 5, stampano un messaggio. Quando però si eseguono questi due frammenti di codice, nei due casi succede qualcosa di molto diverso. nel caso dell'enunciato if, l'output è semplicemente “Ciao, mondo.”; per l'enunciato while, invece, “Ciao, mondo.” viene ripetuto cinque volte!

Date un'occhiata ai diagrammi di flusso per i due frammenti di codice, nelle [Figure 2.9](#) e [2.10](#), per vedere che cosa succede.



**Figura 2.9** - Il diagramma di flusso per l'enunciato if.



**Figura 2.10** - Il diagramma di flusso per l'enunciato while.

Il codice dell'enunciato if verifica la condizione e stampa Ciao mondo solo una volta, se quella condizione è vera. Il codice del ciclo while invece, lo stamperà cinque volte; si ferma dopo averlo stampato cinque volte perché l'intero nella variabile spam viene incrementato di una unità al termine di ciascuna iterazione del ciclo, il che significa che il ciclo viene eseguito cinque volte prima che `spam < 5` diventi False.

Nel ciclo while, la condizione viene sempre verificata all'inizio di ciascuna iterazione (in altre parole, ogni volta che il ciclo viene eseguito). Se la condizione è True, la clausola viene eseguita e poi la condizione viene verificata nuovamente. La prima volta che la condizione risulta False, la clausola while viene saltata.

### ***Un ciclo while irritante***

Ecco un piccolo programma d'esempio che continuerà a chiedervi di scrivere, alla lettera, il tuo nome. Selezionate File > New File per aprire una nuova finestra di file editor, inserite il codice seguente e salvate il file come `yourName.py`:

```

❶ nome = ''
❷ while nome != 'il tuo nome':
    print('Per favore, scrivi il tuo nome.')
    nome = input()
❸ print('Grazie!')

```

Come prima cosa, il programma imposta la variabile nome a una stringa vuota ❶. In questo modo la

condizione `nome != 'il tuo nome'` risulterà True e l'esecuzione del programma entrerà nella clausola del ciclo while ❷.

Il codice all'interno di questa clausola chiede all'utente di scrivere *il tuo nome*, che viene assegnato alla variabile `nome` ❸. Poiché questa è l'ultima riga del blocco, l'esecuzione ritorna all'inizio del ciclo while e valuta di nuovo la condizione. Se il valore nella variabile `nome` non è uguale alla stringa '*il tuo nome*', la condizione è True e l'esecuzione entra nuovamente nella clausola while.

Una volta però che l'utente scriva *il tuo nome*, la condizione del ciclo while diventerà '*il tuo nome* != '*il tuo nome*', il cui valore è False.

La condizione dunque ora è False e l'esecuzione del programma, anziché rientrare nella clausola del ciclo while, salta oltre e continua a eseguire il resto del programma ❹.

La Figura 2.11 presenta un diagramma di flusso per il programma *yourName.py*.

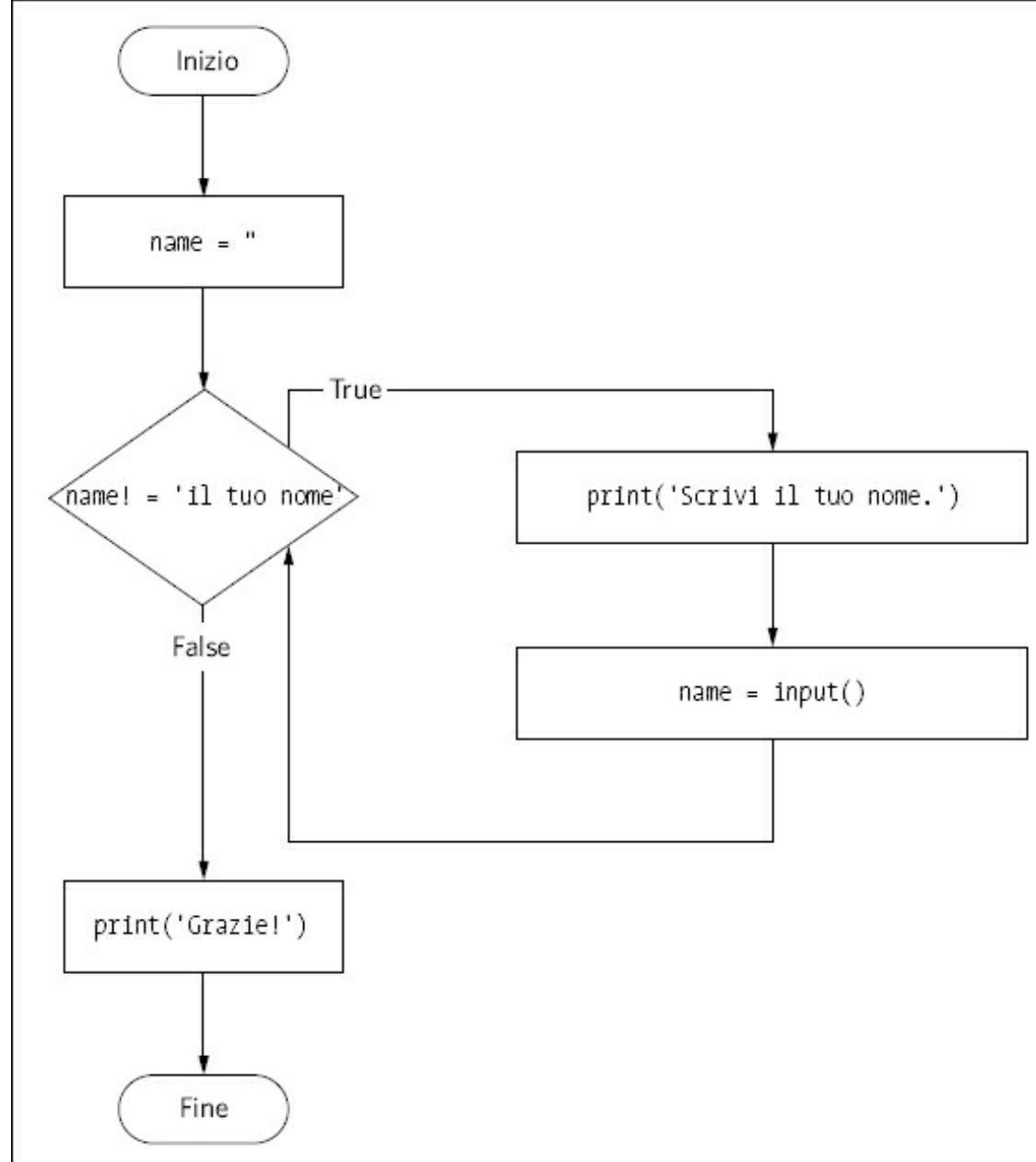


Figura 2.11 - Un diagramma di flusso del programma *yourName.py*.

Ora, vediamo *yourName.py* in azione. Premete F5 per eseguirlo e inserite qualcosa di diverso da *il tuo nome* un po' di volte, prima di dare al programma quello che vuole.

Per favore, scrivi il tuo nome.

**A1**

Per favore, scrivi il tuo nome.

**Alberto**

Per favore, scrivi il tuo nome.

**%#@#%\*(^&!!!**

Per favore, scrivi il tuo nome.

**il tuo nome**

Grazie!

Se non scrivete mai *il tuo nome*, la condizione del ciclo while non diventerà mai False, e il programma continuerà a chiedervelo per sempre.

Qui, la chiamata `input()` consente all'utente di inserire la stringa giusta per far andare avanti il programma. In altri programmi, la condizione potrebbe in effetti non cambiare mai, e questo può essere un problema.

Vediamo allora come si possa uscire da un ciclo while.

## Enunciati break

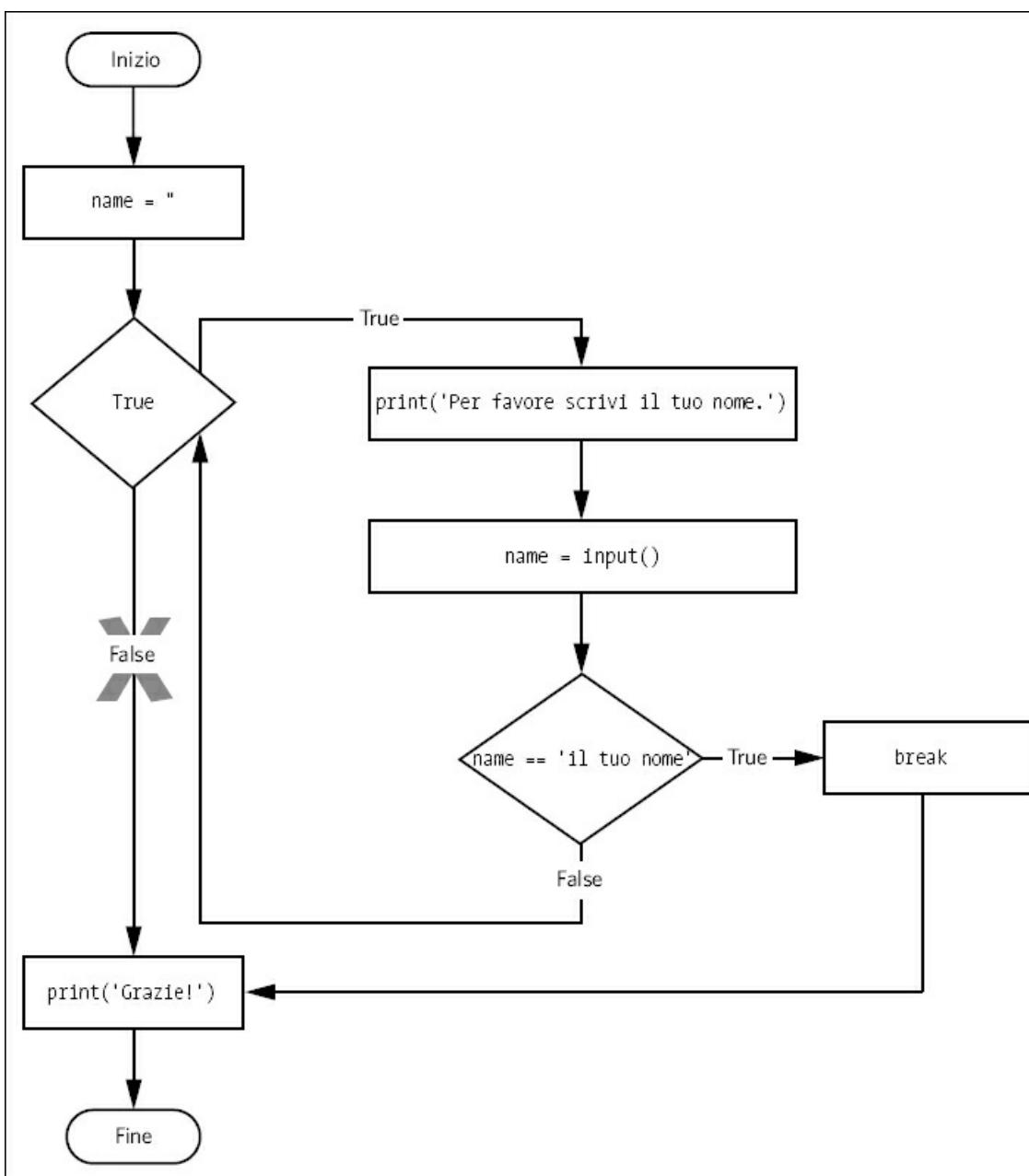
Esiste una scorciatoia per fare in modo che l'esecuzione di un programma esca prima del tempo da una clausola di ciclo while. Se l'esecuzione raggiunge un enunciato `break` (ovvero "interrompi"), esce immediatamente dalla clausola del ciclo. Nel codice, un enunciato `break` contiene semplicemente la parola chiave `break`.

Semplice, vero? Ecco un programma che fa la stessa cosa del programma precedente, ma usa un enunciato `break` per uscire dal ciclo. Inserite il codice seguente e salvate il file con il nome `yourName2.py`.

```
❶ while True:  
    print('Per favore, scrivi il tuo nome.')  
❷    nome = input()  
❸    if nome == 'il tuo nome':  
❹        break  
❺    print('Grazie!')
```

La prima riga ❶ crea un ciclo infinito; è un ciclo while la cui condizione è sempre `True`. (L'espressione `True`, in fin dei conti, viene sempre valutata `True`.) L'esecuzione del programma entra sempre nel ciclo e ne esce solo quando viene eseguito un enunciato `break`. (Un ciclo infinito da cui non si esce mai è un tipico errore di programmazione.)

Proprio come prima, il programma chiede all'utente di scrivere *il tuo nome* ❷. Ora, però, mentre l'esecuzione è ancora all'interno del ciclo while, viene eseguito un enunciato `if` ❸ per verificare se `nome` è uguale a *il tuo nome*. Se questa condizione è `True`, viene eseguito l'enunciato `break` ❹ e l'esecuzione esce dal ciclo per andare a `print('Grazie!')` ❺. Altrimenti, la clausola dell'enunciato `if` con l'enunciato `break` viene saltata, il che porta l'esecuzione alla fine del ciclo while. A questo punto l'esecuzione del programma salta di nuovo all'inizio dell'enunciato `while` ❶ per controllare di nuovo la condizione. Dato che questa condizione è semplicemente il valore Booleano `True`, l'esecuzione entra nel ciclo per chiedere all'utente di scrivere *il tuo nome*. La Figura 2.12 mostra il diagramma di flusso di questo programma.



**Figura 2.12** - Il diagramma di flusso del programma `yourName2.py` con un ciclo infinito. Notate che il percorso indicato con la X logicamente non verrà mai imboccato, perché la condizione del ciclo è sempre True.

Eseguite `yourName.py` e inserite lo stesso testo che avete inserito per `yourName.py`. Il programma riscritto risponderà nello stesso modo dell'originale.

## Enunciati continue

Come gli enunciati `break`, gli enunciati `continue` vengono utilizzati all'interno dei cicli. Quando raggiunge un enunciato `continue`, l'esecuzione del programma salta immediatamente di nuovo all'inizio del ciclo e valuta nuovamente la condizione del ciclo. Questo è anche quello che succede quando l'esecuzione raggiunge la fine del ciclo.)

### Intrappolati in un ciclo infinito?

Se vi dovesse mai capitare di eseguire un programma con un "baco" che lo fa finire in un ciclo infinito, premete Ctrl-C. Questo invierà al programma un errore `KeyboardInterrupt` e farà in modo che si fermi immediatamente. Per provarlo, create un semplice ciclo infinito nel file editor e salvatelo con il nome `infiniteloop.py`.

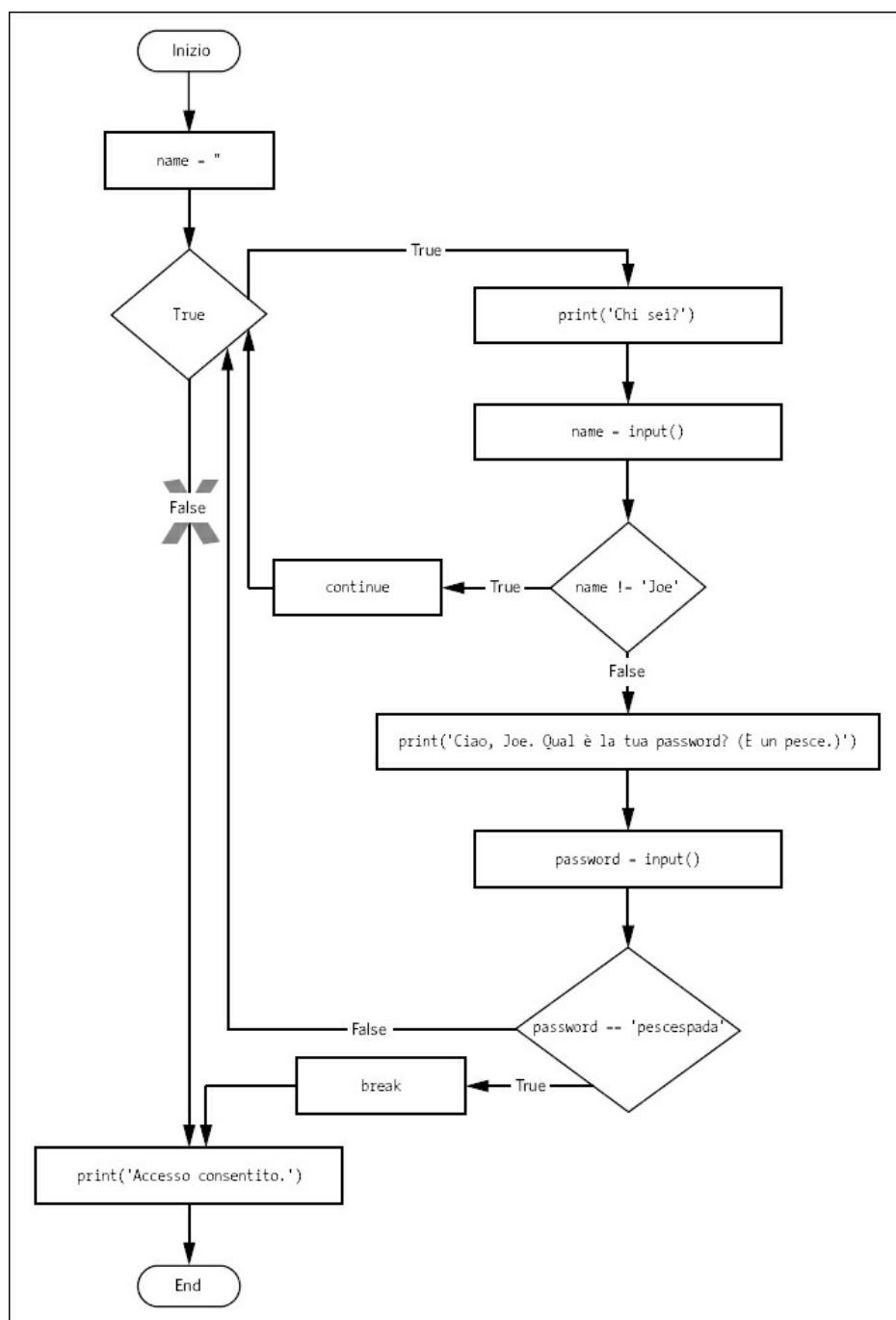
```
while True:  
    print('Ciao mondo!')
```

Quando eseguite il programma, continuerà a stampare Ciao mondo! sullo schermo all’infinito, perché la condizione dell’enunciato while è sempre True. Nella finestra della shell interattiva IDLE, esistono solamente due modi per chiudere questo programma: premere Ctrl-C oppure selezionare **Shell > Restart Shell** dal menu. Ctrl-C è un comando comodo se volete terminare il programma immediatamente, anche se non si è inceppato in un ciclo infinito.

Proviamo a usare `continue` per scrivere un programma che chiede un nome e una password. Inserite il codice seguente in una nuova finestra del file editor e salvate il programma con il nome *swordfish.py*.

```
while True:  
    print('Chi sei?')  
    nome = input()  
   ❶    if nome != 'Joe':  
   ❷        continue  
   ❸    print('Ciao, Joe. Qual è la tua password? (È un pesce.)')  
   ❹    password = input()  
   ❺    if password == 'pescespada':  
        break  
   ❻    print('Accesso consentito.')
```

Se l’utente inserisce un nome qualsiasi che non sia Joe ❶, l’enunciato `continue` ❷ fa sì che l’esecuzione del programma salti indietro all’inizio del ciclo. Quando calcola nuovamente il valore della condizione, l’esecuzione entra sempre nel ciclo, perché la condizione è semplicemente il valore True. Una volta che viene superato l’enunciato `if`, all’utente viene chiesta una password ❸. Se la password inserita è `pescespada`, viene eseguito l’enunciato `break` ❹ e l’esecuzione salta all’esterno del ciclo `while` per stampare Accesso consentito ❺. Altrimenti l’esecuzione continua alla fine del ciclo `while`, da dove salta di nuovo all’inizio del ciclo. La [Figura 2.13](#) mostra il diagramma di flusso per questo programma.



**Figura 2.13** - Il diagramma di flusso del programma `swordfish.py`. Il percorso indicato con la X logicamente non verrà mai imboccato, perché la condizione del ciclo è sempre True.

## Valori “veri” e valori “falsi”

In altri tipi di dati esistono alcuni valori che le condizioni considerano equivalenti a True e False. Quando vengono usati nelle condizioni, 0, 0.0 e " (la stringa vuota) sono considerati False, mentre qualsiasi altro valore è considerato True. Per esempio, esaminate questo programma:

```

nome = ''
while not nome:①
    print('Inserisci il tuo nome:')
    nome = input()
print('Quanti ospiti avrai?')
numDiOspiti = int(input())
if numDiOspiti:②
    print('Assicurati di avere abbastanza spazio per tutti i tuoi ospiti.')③
print('Fatto')

```

Se l'utente inserisce una stringa vuota al posto del nome, la condizione dell'enunciato while risulterà True ①, e il programma continuerà a chiedere un nome. Se il valore di numDiOspiti non è 0 ②, la condizione è considerata True, e il programma stamperà un promemoria per l'utente ③.

Si sarebbe potuto scrivere `not nome != "`, anziché `not nome`, e scrivere `numDiOspiti != 0` invece di `numDiOspiti`, ma l'uso dei valori “veri” e “falsi” in genere può rendere più agevole la lettura del codice.

Eseguite il programma e fornitegli qualche input. Fino a che non dichiarerete di essere Joe, non dovrebbe chiedervi una password; poi, se inserite la password corretta, dovrebbe concludersi.

Chi sei?

**Sto bene, grazie. E tu chi sei?**

Chi sei?

**Joe**

Ciao, Joe. Qual è la tua password? (È un pesce.)

**Mary**

Chi sei?

**Joe**

Ciao, Joe. Qual è la tua password? (È un pesce.)

**pescespada**

Accesso consentito

## Cicli for e la funzione range()

Il ciclo while continua a essere eseguito per tutto il tempo in cui la sua condizione è True (cioè “mentre” la condizione è vera, il che spiega il nome del ciclo); ma se invece si volesse eseguire un blocco di codice solo per un certo numero di volte? Lo si può fare con un enunciato di ciclo for e con la funzione `range()`. Nel codice, un enunciato di ciclo for ha una forma del tipo `for i in range(5)`, e comprende sempre gli elementi seguenti:

- la parola chiave `for`;
- un nome di variabile;
- la parola chiave `in`;
- la chiamata al metodo `range()` a cui possono essere passati fino a tre interi;
- un segno di due punti;
- a partire dalla riga successiva, un blocco di codice rientrato (chiamato clausola for).

Proviamo a creare un nuovo programma, che chiameremo `fiveTimes.py`, per vedere un ciclo for in azione.

```
print('Il mio nome è')
for i in range(5):
    print('Jimmy Cinque Volte (' + str(i) + ')')
```

Il codice nella clausola del ciclo for viene eseguito cinque volte. La prima volta, la variabile `i` ha il valore 0. La chiamata a `print()` nella clausola stamperà Jimmy Cinque Volte (0). Quando Python completa un’iterazione di tutto il codice all’interno della clausola del ciclo for, l’esecuzione torna all’inizio del ciclo e l’enunciato `for` aumenta `i` di una unità. Questo è il motivo per cui `range(5)` dà come risultato cinque iterazioni della clausola, con `i` impostata a 0, poi a 1, poi a 2, poi a 3 e infine a 4. La variabile `i`

arriva fino all'intero (escluso) passato a `range()`. La Figura 2.14 mostra un diagramma di flusso del programma `fiveTimes.py`.

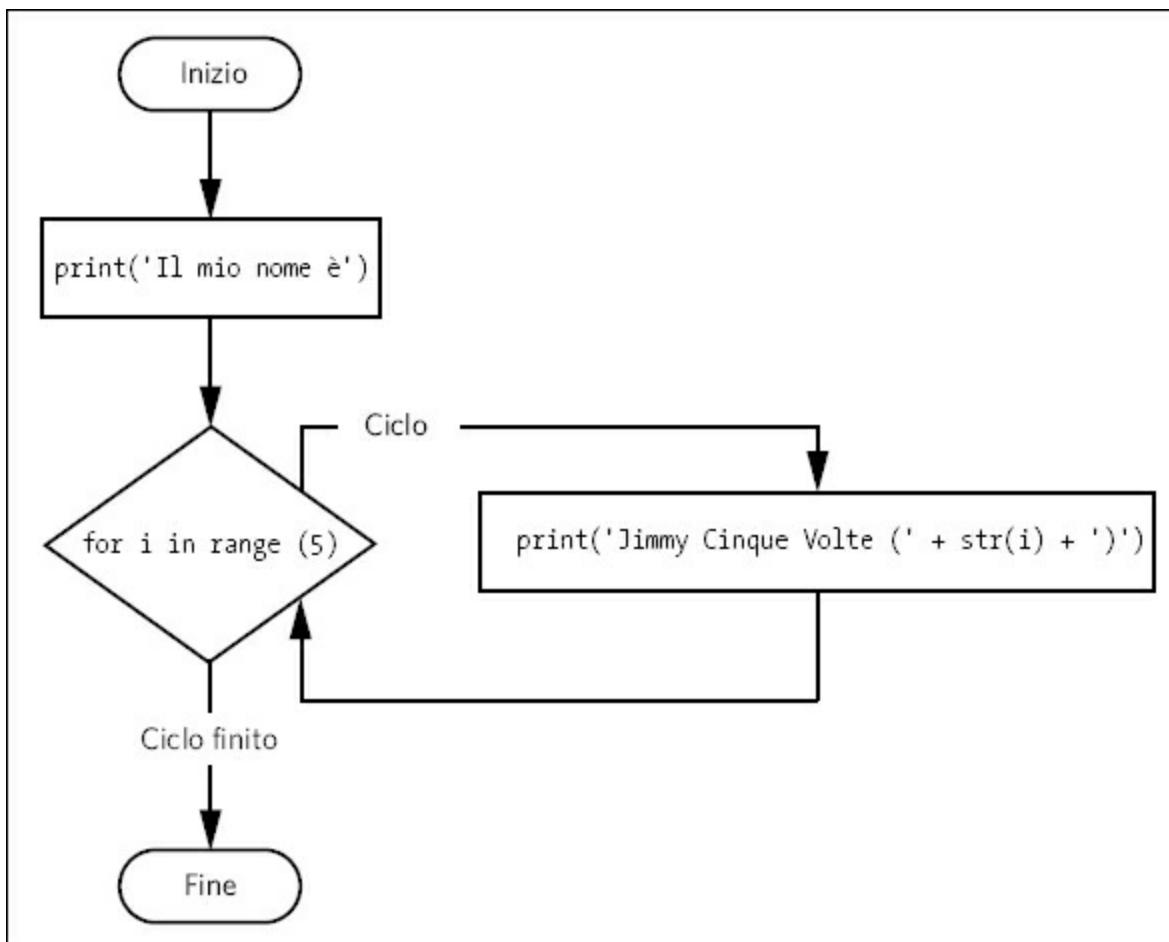


Figura 2.14 - Il diagramma di flusso per `fiveTimes.py`.

Quando si esegue questo programma, dovrebbe stampare per cinque volte Jimmy Cinque Volte seguito dal valore di `i`, prima di uscire dal ciclo `for`.

```
Il mio nome è
Jimmy Cinque Volte (0)
```

## NOTA

Potete usare gli enunciati `break` e `continue` anche all'interno dei cicli `for`. L'enunciato `continue` farà continuare l'esecuzione al valore successivo del contatore del ciclo `for`, come se l'esecuzione del programma avesse raggiunto la fine del ciclo e fosse tornata all'inizio. In effetti si possono usare gli enunciati `continue` e `break` solo all'interno dei cicli `while` e `for`. Se tentate di utilizzarli altrove, Python vi presenterà un messaggio d'errore.

Come altro esempio di ciclo `for`, possiamo partire ricordando il matematico Karl Friedrich Gauss.

Quando Gauss era ancora un ragazzo, un insegnante propose a lui e ai suoi compagni di classe un compito che pensava fosse impegnativo: sommare tutti i numeri da 0 a 100. Gauss escogitò un trucco molto astuto che gli permise di trovare la risposta in pochi secondi, ma voi potete scrivere un programma Python con un ciclo `for` che esegua questo calcolo per voi.

```
❶ total = 0
❷ for num in range(101):
❸     total = total + num
❹ print(total)
```

Il risultato deve essere 5050. Quando il programma inizia, la variabile `total` viene impostata a 0 ❶. Il ciclo `for` ❷ poi esegue `total = total + num` ❸ per 100 volte. Quando il ciclo ha finito tutte le sue 100 iterazioni, a `total` è stato sommato ogni intero da 0 a 100. A questo punto, viene stampato sullo schermo `total` ❹. Anche sui computer più lenti, questo programma viene eseguito in meno di un secondo. (Il giovane Gauss ha pensato che c'erano 50 coppie di numeri la cui somma era 100: 1 + 99, 2 + 98, 3 + 97 e così via, fino a 49 + 51. Poiché  $50 \times 100 = 5000$ , quando si somma il 50 centrale, la somma di tutti i numeri da 0 a 100 risulta 5050. Ragazzo brillante, quel Gauss!)

## ***Un ciclo while equivalente***

Si può usare un ciclo `while` per fare la stessa cosa che fa un ciclo `for`: ma i cicli `for` sono semplicemente più concisi. Proviamo a riscrivere *fiveTimes.py* usando un ciclo `while` equivalente a un ciclo `for`.

```
print('Il mio nome è')
i = 0
while i < 5:
    print('Jimmy Cinque Volte (' + str(i) + ')')
    i = i + 1
```

Se eseguite questo programma, l'output sarà lo stesso del programma *fiveTimes.py*, che usa un ciclo `for`.

## ***Gli argomenti di `range()`: start, stop e step***

Alcune funzioni possono essere chiamate con più argomenti, separati fra loro da una virgola, e `range()` è una di queste. In questo modo è possibile modificare l'intero passato a `range()`, generando una successione qualsiasi di interi, e si può indicare anche un numero di inizio diverso da zero.

```
for i in range(12, 16):
    print(i)
```

Il primo argomento indica il punto di inizio della variabile del ciclo `for`; il secondo indica il numero a cui il ciclo deve fermarsi (senza usare quel numero).

12  
13  
14  
15

La funzione `range()` può essere chiamata anche con tre argomenti. I primi due saranno i valori di `start` e `stop`, cioè di **inizio** e **fine**, il terzo sarà l'`argomento step`, ovvero il **passo** del ciclo. Il passo indica il

valore di cui deve essere aumentata la variabile a ogni iterazione.

```
for i in range(0, 10, 2):
    print(i)
```

Quindi chiamando `range(0, 10, 2)` avremo la successione dei numeri da zero a otto, con passo due.

```
0
2
4
6
8
```

La funzione `range()` è molto flessibile, per quanto riguarda la successione di numeri che produce per i cicli `for`. Per esempio, si può addirittura usare un numero negativo per l'argomento `passo`, in modo che il ciclo `for` dia una successione decrescente, anziché crescente.

```
for i in range(5, -1, -1):
    print(i)
```

L'esecuzione di un ciclo `for` che stampi `i` con `range(5, -1, -1)` stamperà i numeri da cinque a zero.

```
5
4
3
2
1
0
```

## Importare moduli

Tutti i programmi Python possono chiamare un insieme fondamentale di funzioni, chiamate **funzioni interne** (*built-in functions*), fra cui `print()`, `input()` e `len()`, che abbiamo già visto. Python però dispone anche di un insieme di moduli, che costituiscono la **libreria standard**. Ciascun modulo è un programma Python che contiene un gruppo di funzioni, fra loro collegate, e che può essere incorporato nei programmi. Per esempio, il modulo `math` contiene funzioni di tipo matematico, il modulo `random` contiene funzioni relative alla generazione di numeri casuali (`random`) e così via.

Per poter utilizzare le funzioni contenute in un modulo, bisogna prima **importare il modulo** con un enunciato `import`. Nel codice, un enunciato `import` è costituito da:

- la parola chiave `import`;
- il nome del modulo;
- facoltativamente, altri nomi di moduli, purché siano separati fra loro da virgole.

Una volta importato un modulo, si possono usare tutte le sue funzioni. Proviamo con il modulo `random`, che permette di utilizzare la funzione `random.randint()`.

Inserite nel file editor questo codice e salvatelo con il nome `printRandom.py`:

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

Quando eseguite questo programma, l'output sarà qualcosa di questo genere:

```
4  
1  
8  
4  
1
```

La funzione `random.randint()` dà come valore un intero casuale compreso fra due interi che le vengono passati come argomenti. Poiché `randint()` è contenuta nel modulo `random`, dovete prima scrivere `random`, davanti al nome della funzione, per dire a Python di cercare quella funzione all'interno del modulo `random`.

Ecco un esempio di enunciato `import` che importa quattro moduli diversi:

```
import random, sys, os, math
```

Ora possiamo usare tutte le funzioni contenute in questi quattro moduli. Vedremo altro in proposito nel seguito del libro.

### ***Enunciati from import***

Una forma alternativa dell'enunciato `import` è costituita dalla parola chiave `from`, seguita dal nome del modulo, dalla parola chiave `import` e da un asterisco; per esempio, `from random import *`.

Con questa forma dell'enunciato `import`, le chiamate alle funzioni contenute in `random` non avranno bisogno del prefisso `random.`; l'uso del nome completo però rende il codice più leggibile, perciò è meglio usare la forma normale dell'enunciato `import`.

## **Conclusione prematura di un programma con `sys.exit()`**

L'ultimo concetto relativo al controllo del flusso di cui dobbiamo parlare è **come terminare il programma**. Questo succede sempre se l'esecuzione del programma raggiunge la fine delle istruzioni. Tuttavia, è possibile far sì che il programma termini, o "esca", chiamando la funzione `sys.exit()`. Questa funzione è contenuta nel modulo `sys`, perciò dovete importare `sys` perché il programma possa usarla. Aprite una nuova finestra del file editor e inserite il codice seguente, poi salvatelo con il nome `exitExample.py`:

```
import sys  
  
while True:  
    print('Scrivi exit per uscire.')  
    response = input()  
    if response == 'exit':  
        sys.exit()  
    print('Hai scritto ' + response + '.')
```

Eseguite questo programma in IDLE. Il programma ha un ciclo infinito senza alcun enunciato `break` al suo interno. L'unico modo in cui il programma finirà è se l'utente inserisce `exit`, provocando la chiamata di `sys.exit()`. Quando `response` è uguale a `exit`, il programma termina. Poiché la variabile `response` è impostata dalla funzione `input()`, l'utente deve inserire `exit` per poter fermare il programma.

# Riepilogo

Utilizzando espressioni che vengono valutate True o False (chiamate anche **condizioni**), potete scrivere programmi che prendono **decisioni** su quale codice eseguire e quale saltare. Potete anche eseguire **blocchi di codice** più volte in un **ciclo**, fintanto che una data condizione ha valore True. Gli enunciati break e continue sono utili se si ha bisogno di uscire da un ciclo o di saltare di nuovo all'inizio.

Questi **enunciati di controllo del flusso** vi permetteranno di scrivere programmi molto più intelligenti. Esiste un altro tipo di controllo del flusso che si può ottenere scrivendo le proprie funzioni, e questo è l'argomento del prossimo capitolo.

## Un po' di pratica

1. Quali sono i due valori del tipo di dati Booleano? Come si scrivono?
2. Quali sono i tre operatori Booleani?
3. Scrivete le tavole di verità di ciascuno degli operatori Booleani (cioè tutte le possibili combinazioni di valori Booleani per l'operatore e qual è il valore che danno come risultato).
4. Qual è il valore che hanno le espressioni seguenti?

( $5 > 4$ ) and ( $3 == 5$ )  
not ( $5 > 4$ )  
( $5 > 4$ ) or ( $3 == 5$ )  
not (( $5 > 4$ ) or ( $3 == 5$ ))  
(True and True) and (True == False)  
(not False) or (not True)

5. Quali sono i sei operatori di confronto?
6. Qual è la differenza fra l'operatore di uguaglianza e l'operatore di assegnazione?
7. Spiegate che cos'è una condizione e dove la usereste.
8. Identificate i tre blocchi in questo codice:

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
    else:
        print('ham')
    print('spam')
print('spam')
```

9. Scrivete codice che stampi Hello se in spam è memorizzato 1, stampi Howdy se in spam è memorizzato 2, e stampi Greetings! se in spam è memorizzato qualcosa di diverso.
10. Quali tasti potete premere, se un programma è bloccato in un ciclo infinito?
11. Qual è la differenza fra break e continue?
12. Qual è la differenza fra range(10), range(0, 10) e range(0, 10, 1) in un ciclo for?
13. Scrivete un breve programma che stampi i numeri da 1 a 10 utilizzando un ciclo for. Poi scrivete un programma equivalente che stampi i numeri da 1 a 10 utilizzando un ciclo while.
14. Se avete una funzione bacon() in un modulo chiamato spam, come la chiamereste dopo aver importato spam?

**Credito extra:** Cercate in Internet le funzioni `round()` e `abs()` e scoprirete che cosa fanno. Sperimentatele nella shell interattiva.

# Funzioni

Concete già le funzioni `print()`, `input()` e `len()` perché le abbiamo incontrate nei capitoli precedenti. Python mette a disposizione parecchie **funzioni interne** come queste, ma potete anche scrivere le vostre funzioni. Una **funzione** è come un piccolo **programma** all'interno di un programma.

Per capire meglio che cosa sono le **funzioni**, creiamone una. Scrivete questo programma nel file editor e salvatelo con il nome *helloFunc.py*.

```
❶ def hello():
❷     print('Ciao!')
        print('Ciao!!!')
        print('Ciao a te.')
❸ hello()
    hello()
    hello()
```

La prima riga è un enunciato `def` ❶, che definisce una funzione il cui nome è `hello()`. Il codice nel blocco che segue l'enunciato `def` ❷ è il corpo della funzione. Questo codice viene eseguito quando la funzione viene chiamata, ma non quando la funzione viene definita.

Le righe `hello()` dopo la funzione ❸ sono **chiamate** della funzione. Nel codice, una chiamata di funzione è semplicemente costituita dal nome della funzione seguito da parentesi, eventualmente con alcuni argomenti fra parentesi. Quando l'esecuzione del programma raggiunge queste chiamate, salta alla prima riga della funzione e inizia a eseguire il codice che vi trova. Quando raggiunge la fine della funzione, l'esecuzione torna alla riga che aveva chiamato la funzione e continua quindi a eseguire il codice come prima.

Poiché questo programma chiama `hello()` tre volte, il codice nella funzione `hello()` viene eseguito tre volte. Quando eseguite questo programma, l'output è come questo:

Ciao!!!  
Ciao a te.  
Ciao!  
Ciao!!!  
Ciao a te.  
Ciao!  
Ciao!!!  
Ciao a te.

Una delle finalità principali delle funzioni è quella di **raggruppare codice** che viene eseguito molte volte.

Se non fosse stata definita una funzione, avreste dovuto copiare e incollare ogni volta questo codice, e il programma avrebbe un aspetto come questo:

```
print('Ciao!')  
print('Ciao!!!!')  
print('Ciao a te.')  
print('Ciao!')  
print('Ciao!!!!')  
print('Ciao a te.')  
print('Ciao!')  
print('Ciao!!!!')  
print('Ciao a te.')  
print('Ciao!')
```

In generale, si **evita sempre di duplicare codice**, perché, se mai si dovesse decidere di aggiornarlo (se per esempio si trova un errore che si deve correggere), bisogna ricordarsi di modificare il codice ovunque sia stato copiato.

A mano a mano che la vostra esperienza di programmazione aumenterà, vi troverete spesso a **deduplicare** codice, il che significa togliere di mezzo codice duplicato o copiato e incollato. La deduplicazione rende i programmi più brevi, più facili da leggere e più facili da aggiornare.

## Enunciati def con parametri

Quando chiamate la funzione `print()` o `len()`, le passate dei valori, chiamati **argomenti** in questo contesto, scrivendoli fra parentesi.

Potete anche definire vostre funzioni che accettano argomenti. Scrivete l'esempio seguente nel file editor e salvatelo con il nome `helloFunc2.py`:

```
❶ def hello(nome):  
❷     print('Ciao ' + nome)  
❸ hello('Alice')  
    hello('Bob')
```

Se eseguite questo programma, l'output sarà:

Ciao Alice  
Ciao Bob

La definizione della funzione `hello()` in questo programma ha un **parametro**, chiamato `nome` ❶. Un parametro è una variabile in cui viene memorizzato un argomento, quando viene chiamata una funzione. La prima volta che la funzione `hello()` viene chiamata, le viene passato l'argomento 'Alice' ❸.

L'esecuzione del programma entra nella funzione e la variabile nome viene automaticamente impostata ad 'Alice', che è il nome che viene poi stampato dall'enunciato print() ❷.

Una cosa particolare da notare, a proposito dei parametri, è che il valore memorizzato in un parametro viene dimenticato quando la funzione ritorna. Per esempio, se avete aggiunto print(nome) dopo hello('Bob') nel programma precedente, avreste ottenuto un NameError, perché non esiste più una variabile nome. Questa variabile è stata distrutta dopo il ritorno della chiamata di funzione hello('Bob'), perciò print(nome) farebbe riferimento a una variabile che non esiste più.

È un po' come quello che succede alle variabili di un programma, che vengono "dimenticate", quando il programma giunge a conclusione. Dirò di più sul perché di questa cosa nel seguito del capitolo, quando parleremo dell'ambito locale di una funzione.

## Valori di ritorno ed enunciati return

Quando chiamate la funzione len() e le passate un argomento come 'Ciao', la chiamata di funzione calcola come valore l'intero 4, che è la lunghezza della stringa passata. In generale, il valore che una funzione calcola è chiamato **valore di ritorno** della funzione (o **valore restituito** dalla funzione).

Quando create una funzione con l'enunciato def, specificate quale debba essere il valore restituito con un enunciato return. Un enunciato return è costituito da:

- la parola chiave return;
- il valore o l'espressione che la funzione deve restituire.

Quando in un enunciato return si usa un'espressione, il valore restituito è quello a cui viene valutata quell'espressione. Per esempio, il programma seguente definisce una funzione che restituisce una stringa diversa a seconda del numero che le viene passato come argomento.

Scrivete questo codice nel file editor e salvatelo con il nome *magic8Ball.py*.

```
❶ import random

❷ def getAnswer(answerNumber):
❸     if answerNumber == 1:
❹         return 'È sicuro'
    elif answerNumber == 2:
        return 'Proprio così'
    elif answerNumber == 3:
        return 'Sì'
    elif answerNumber == 4:
        return 'Risposta incerta, riprova'
    elif answerNumber == 5:
        return 'Chiedi di nuovo più tardi'
    elif answerNumber == 6:
        return 'Concentrati e chiedi di nuovo'
    elif answerNumber == 7:
        return 'La mia risposta è no'
    elif answerNumber == 8:
        return 'La prospettiva non è molto buona'
    elif answerNumber == 9:
        return 'Molto dubioso'

❺ r = random.randint(1, 9)
❻ fortune = getAnswer(r)
❼ print(fortune)
```

Quando il programma viene lanciato, per prima cosa importa il modulo `random` ❶. Poi viene definita la funzione `getAnswer()` ❷. Poiché la funzione qui viene definita (e non chiamata) l'esecuzione salta il codice della definizione. Poi viene chiamata la funzione `random.randint()` con due argomenti, 1 e 9 ❸. La funzione restituisce un intero casuale fra 1 e 9 (compresi questi due numeri) e il valore è memorizzato in una variabile `r`.

La funzione `getAnswer()` viene chiamata con `r` come argomento ❹. L'esecuzione del programma passa all'inizio della funzione `getAnswer()` ❺ e il valore `r` è memorizzato in un parametro `answerNumber`. Poi, a seconda del valore in `answerNumber`, la funzione restituisce uno dei tanti possibili valori stringa. L'esecuzione del programma torna alla riga dove si trovava la chiamata alla funzione `getAnswer()` ❻. La stringa restituita viene assegnata a una variabile `fortune`, che poi viene passata a una chiamata di `print()` ❻ e viene stampata sullo schermo.

Notate che è possibile passare i valori restituiti come argomento a un'altra chiamata di funzione, e così le tre righe seguenti:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

possono essere ridotte a una sola equivalente:

```
print(getAnswer(random.randint(1, 9)))
```

Ricordate: le espressioni sono formate da valori e operatori. Una chiamata di funzione può essere utilizzata all'interno di un'espressione perché viene valutata e il risultato è il suo valore di ritorno.

## Il valore None

In Python esiste un valore, `None`, che rappresenta l'**assenza di un valore**. `None` è l'unico valore del tipo di dati `NoneType`. (Altri linguaggi di programmazione chiamano questo valore `null`, `nil` o `undefined`.) Come i valori Booleani `True` e `False`, `None` deve essere scritto con l'iniziale maiuscola.

Questo "valore senza valore" può essere utile quando si rende necessario memorizzare qualcosa che non deve essere confuso con un valore reale in una variabile. Un caso in cui si usa `None` è come valore di ritorno di `print()`. La funzione `print()` visualizza del testo su schermo, ma non deve restituire qualcosa come fanno `len()` o `input()`. Dato però che ogni chiamata di funzione deve avere un valore di ritorno, `print()` restituisce `None`. Per vederlo in azione, inserite nella shell interattiva quanto segue:

```
>>> spam = print('Ciao!')
Ciao!
>>> None == spam
True
```

Dietro le quinte, Python aggiunge `return None` alla fine di ogni definizione di funzione che non contenga un enunciato `return`. In modo simile, un ciclo `while` o `for` finisce implicitamente con un enunciato `continue`. Inoltre, se usate un enunciato `return` senza un valore (cioè scrivendo solo la parola chiave `return`), viene restituito `None`.

## Argomenti per parola chiave e `print()`

Nella maggior parte dei casi, gli argomenti sono identificati in base alla loro posizione nella chiamata della funzione. Per esempio, `random.randint(1, 10)` è diverso da `random.randint(10, 1)`. La chiamata di funzione `random.randint(1, 10)` restituirà un intero scelto a caso fra 1 e 10, perché il primo argomento è l'estremo inferiore dell'intervallo e il secondo argomento è l'estremo superiore; `random.randint(10, 1)` provoca invece un errore.

Gli **argomenti per parola chiave** invece sono identificati dalla parola chiave che viene messa loro davanti nella chiamata di funzione. Questi argomenti vengono usati spesso per **parametri facoltativi**. Per esempio, la funzione `print()` ha i parametri facoltativi `end` e `sep` per precisare che cosa debba essere stampato alla fine dei suoi argomenti e fra un argomento e l'altro (per separarli), rispettivamente. Se eseguite il programma seguente:

```
print('Ciao')
print('mondo')
```

l'output sarà:

```
Ciao
mondo
```

Le due stringhe compaiono su righe diverse, perché la funzione `print()` aggiunge automaticamente un carattere di "a capo" (*newline*) alla fine della stringa che le viene passata.

Potete però impostare l'argomento per parola chiave `end` in modo da avere un comportamento diverso. Per esempio, se il programma fosse questo:

```
print('Ciao', end="")
print('mondo')
```

l'output sarebbe

```
Ciaomondo
```

Questa volta l'output è stampato su un'unica riga perché non viene più stampato un a capo dopo 'Ciao': viene stampata invece una stringa vuota.

Questo è utile se si vuole disabilitare l'a capo che viene aggiunto alla fine di ogni chiamata alla funzione `print()`.

Analogamente, se si passano più valori stringa a `print()`, la funzione li separerà automaticamente con uno spazio singolo. Scrivete nella shell interattiva:

```
>>> print('gatti', 'cani', 'topi')
gatti cani topi
```

Ma potreste sostituire la stringa di separazione implicita passando l'argomento `sep`. Inserite nella shell interattiva:

```
>>> print('gatti', 'cani', 'topi', sep=',')
gatti,cani,topi
```

Potete aggiungere argomenti per parola chiave anche alle funzioni che scrivete voi, ma per poterlo fare dovete conoscere i tipi di dati lista e dizionario, di cui parleremo nei prossimi due capitoli.

Per ora, basta sappiate che alcune funzioni hanno argomenti per parola chiave facoltativi che possono essere specificati quando la funzione viene chiamata.

## Ambito locale e globale

Parametri e variabili assegnati a una funzione chiamata si dice che hanno un **ambito** (o **raggio d'azione**, in inglese **scope**) **locale** a quella funzione. Le variabili assegnate all'esterno di tutte le funzioni si dice che hanno invece un **ambito** (o *raggio d'azione*) **globale**. Una variabile con ambito locale è una **variabile locale**; una variabile con ambito globale è chiamata anche **variabile globale**. Una variabile può essere solo di un tipo: non può essere contemporaneamente locale e globale.

Pensate l'ambito come una sorta di contenitore per variabili. Quando un ambito viene distrutto, tutti i valori conservati nelle variabili di quell'ambito vengono dimenticati. Esiste solo un ambito globale, e viene creato quando inizia il programma. Quando il programma termina, l'ambito globale è distrutto e tutte le sue variabili sono dimenticate. Altrimenti, alla successiva esecuzione del programma, le variabili ricorderebbero il valore che avevano al termine dell'esecuzione precedente. Un ambito locale si crea ogni volta che viene chiamata una funzione. Qualsiasi variabile che riceva un'assegnazione in quella funzione esiste solo entro l'ambito locale. Quando la funzione ritorna, l'ambito locale viene distrutto e quelle variabili vengono dimenticate. Alla prossima chiamata della funzione, le variabili locali non ricorderanno i valori che erano memorizzati al loro interno alla precedente chiamata della funzione.

Il concetto di ambito è importante per vari motivi.

- Il codice nell'ambito globale non può usare variabili locali.
- Un ambito locale invece può accedere alle variabili globali.
- Il codice nell'ambito locale di una funzione non può usare variabili che si trovino in qualche altro ambito locale.
- Potete usare lo stesso nome per variabili diverse se si trovano in ambiti diversi. In altre parole, può esistere una variabile locale che si chiama `spam` e una variabile globale che si chiama a sua volta `spam`.

Il motivo per cui Python ha ambiti diversi, anziché un solo ambito globale, è che quando le variabili vengono modificate dal codice in una particolare chiamata a una funzione, la funzione interagisce con il resto del programma solo attraverso i suoi parametri e il valore di ritorno. Questo riduce il numero delle righe di codice che possono provocare un errore. Se il vostro programma non contenesse altro che variabili globali e avesse un baco a causa di una variabile impostata a un valore sbagliato, sarebbe difficile capire dove sia stato impostato quel valore sbagliato. Potrebbe essere stato impostato in qualsiasi punto del programma, e il programma potrebbe essere lungo centinaia o migliaia di righe. Ma se l'errore è provocato da una variabile locale con un valore sbagliato, sapete subito che solo il codice in quella particolare funzione può averlo impostato in modo errato.

L'uso di variabili globali in un programma di piccole dimensioni è accettabile, ma è una cattiva abitudine fare affidamento sulle variabili globali quando le dimensioni dei programmi aumentano.

## Le variabili locali non possono essere usate nell'ambito globale

Considerate questo programma, che provoca un errore quando lo eseguite:

```
def spam():
    eggs = 31337
spam()
print(eggs)
```

Se eseguite il programma, l'output sarà simile a questo:

```
Traceback (most recent call last):
  File "C:/test3784.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

L'errore si verifica perché la variabile `eggs` esiste solo nell'ambito locale creato quando è stata chiamata `spam()`. Una volta che l'esecuzione del programma ritorna da `spam`, quell'ambito locale viene distrutto e non esiste più una variabile `eggs`. Perciò, quando il vostro programma cerca di eseguire `print(eggs)`, Python denuncia un errore, dicendo che `eggs` non è definita. Ha senso, se ci pensate: quando l'esecuzione del programma è nell'ambito globale, non esistono ambiti locali, perciò non possono esserci variabili locali. Per questo nell'ambito globale possono essere usate solo variabili globali.

## Gli ambiti locali non possono usare variabili di altri ambiti locali

Si crea un nuovo ambito locale ogni volta che viene chiamata una funzione, anche quando una funzione viene chiamata da un'altra funzione. Guardate per esempio questo programma:

```
❶ def spam():
❷     eggs = 99
❸     bacon()
❹     print(eggs)
❺     def bacon():
❻         ham = 101
❼         eggs = 0
➋     spam()
```

Quando il programma si avvia, viene chiamata la funzione `spam()` ❺ e viene creato un ambito locale. La variabile locale `eggs` ❶ viene impostata a 99. Poi viene chiamata la funzione `bacon()` ❷ e viene creato un secondo ambito locale. Possono esistere più ambiti locali allo stesso tempo. In questo nuovo ambito locale, la variabile locale `ham` è impostata a 101, e viene creata una variabile locale `eggs` – che è diversa da quella nell'ambito locale di `spam()` – a cui viene assegnato il valore 0.

Quando `bacon()` ritorna, l'ambito locale per quella chiamata è distrutto. L'esecuzione del programma continua nella funzione `spam()` e stampa il valore di `eggs` ❸; poiché l'ambito locale per la chiamata a `spam()` qui esiste ancora, la variabile `eggs` è impostata a 99. Questo è ciò che il programma stampa. In sostanza: le variabili locali in una funzione sono completamente separate dalle variabili locali in un'altra funzione.

## Le variabili locali possono essere lette da un ambito locale

Prendete il programma seguente:

```
def spam():
    print(eggs)
```

```
eggs = 42
spam()
print(eggs)
```

Dato che non esiste alcun parametro di nome `eggs` e nessun codice che assegna un valore a `eggs` nella funzione `spam()`, quando `eggs` viene usata in `spam()` Python la considera un riferimento alla variabile globale `eggs`. Per questo, quando viene eseguito il programma, viene stampato 42.

## Variabili locali e globali con lo stesso nome

Per semplificarvi la vita, evitate di usare variabili locali che abbiano lo stesso nome di una variabile globale o di un'altra variabile locale, anche se tecnicamente è perfettamente lecito farlo in Python.

Per vedere che cosa succede, scrivete nel file editor questo codice e salvatelo con il nome di `sameName.py`:

```
❶ def spam():
    eggs = 'spam locale'
    print(eggs)          # stampa 'spam locale'

❷ def bacon():
    eggs = 'bacon locale'
    print(eggs)          # stampa 'bacon locale'
    spam()
    print(eggs)          # stampa 'bacon locale'

❸ eggs = 'globale'
bacon()
print(eggs)          # stampa 'globale'
```

Quando eseguite il programma, dà questo output:

```
bacon locale
spam locale
bacon locale
globale
```

In questo programma esistono effettivamente tre diverse variabili, ma sono tutte chiamate `eggs`, il che confonde un po' le idee. Le variabili sono le seguenti.

- ❶ Una variabile `eggs` che si trova in un ambito locale quando viene chiamata `spam()`.
- ❷ Una variabile `eggs` che si trova in un ambito locale quando viene chiamata `bacon()`.
- ❸ Una variabile `eggs` che si trova nell'ambito globale.

Dato che queste tre distinte variabili hanno lo stesso nome, si fa fatica a seguire quale venga utilizzata in ogni dato momento. Per questo è consigliabile evitare di usare lo stesso nome di variabile in ambiti diversi.

Se dovete **modificare una variabile globale dall'interno di una funzione**, usate l'enunciato global. Se avete una riga come global eggs all'inizio di una funzione, questa dice a Python: "in questa funzione, eggs si riferisce alla variabile globale, perciò non creare una variabile locale con questo nome". Per esempio, scrivete questo codice nel file editor e salvatelo con il nome *sameName2.py*:

```
❶ def spam():
    global eggs
❷     eggs = 'spam'

eggs = 'global'
spam()
print(eggs)
```

Se eseguite questo programma, la chiamata finale a print() stamperà questo:

spam

Poiché eggs è dichiarata global all'inizio di spam() ❶, quando eggs è impostata a 'spam' ❷, questa assegnazione riguarda la variabile eggs con ambito globale. Non viene creata alcuna variabile eggs locale.

Vi sono quattro regole per stabilire se una variabile è locale o globale.

1. Se una variabile viene usata nell'ambito globale (cioè all'esterno di tutte le funzioni), allora è sempre una variabile globale.
2. Se esiste un enunciato global per quella variabile in una funzione, è una variabile globale.
3. Altrimenti, se la variabile è usata in un enunciato di assegnazione nella funzione, è una variabile locale.
4. Ma, se la variabile non è usata in un enunciato di assegnazione, si tratta di una variabile globale.

Per avere un'idea migliore di queste regole, ecco un programma d'esempio. Scrivetelo nel file editor e salvatelo con il nome *sameName3.py*:

```
❶ def spam():
    global eggs
    eggs = 'spam'      # questa è la globale

❷ def bacon():
    eggs = 'bacon'    # questa è una locale

❸ def ham():
    print(eggs)       # questa è la globale

eggs = 42             # questa è la globale
spam()
print(eggs)
```

Nella funzione spam(), eggs è la variabile globale, perché all'inizio della funzione ❶ si trova un enunciato global per eggs.

In bacon(), eggs è una variabile locale, perché in quella funzione vi è un enunciato di assegnazione per quella variabile ❷.

In `ham()` ❸, `eggs` è la variabile globale, perché non vi è enunciato di assegnazione né enunciato global per quella variabile nella funzione.

Se eseguite `sameName3.py`, il risultato sarà:

spam

In una funzione, una variabile sarà o sempre globale o sempre locale. Non esiste la possibilità che il codice in una funzione usi una variabile locale di nome `eggs` e poi nella stessa funzione usi la variabile globale `eggs`.

## NOTA

Se vi capiterà di voler modificare dall'interno di una funzione il valore memorizzato in una variabile globale, dovete usare un enunciato `global` per quella variabile.

Se cercate di usare in una funzione una variabile locale prima di assegnarle un valore, come nel programma seguente, Python denuncerà un errore. Per vederlo, scrivete quanto segue nel file editor e salvatelo con il nome `sameName4.py`.

```
❶ def spam():
    print(eggs)      # ERRORE!
❷     eggs = 'spam locale'

❸ eggs = 'globale'
spam()
```

Se eseguite questo programma, otterrete un messaggio d'errore.

```
Traceback (most recent call last):
  File "C:/test3784.py", line 6, in <module>
    spam()
  File "C:/test3784.py", line 2, in spam
    print(eggs) # ERRORE!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

Questo errore si verifica perché Python vede un enunciato di assegnazione per `eggs` nella funzione `spam()` ❶ e perciò tratta `eggs` come locale. Poiché però `print(eggs)` viene eseguita prima che a `eggs` venga assegnato un valore, la variabile locale `eggs` non esiste. Python in tal caso non ripiega sull'uso della variabile globale `eggs` ❷.

## Funzioni come “scatole nere”

Spesso, tutto quello che serve sapere di una funzione sono i suoi input (i parametri) e il valore di output; non interessa in modo particolare come agisce il codice effettivo della funzione. Quando si pensa a una funzione in questo modo “ad alto livello”, si dice che si tratta la funzione come una “scatola nera”.

L'idea è fondamentale per la programmazione moderna. In capitoli successivi di questo libro vedremo vari moduli con funzioni che sono state scritte da altri. Potete dare un'occhiata al codice sorgente, se siete curiosi, ma non è necessario sapere come sono fatte internamente queste funzioni per poterle usare. Poiché poi si consiglia di scrivere funzioni senza variabili globali, di solito non ci si

## Gestione delle eccezioni

Al momento, se si presenta un **errore**, ovvero una **eccezione**, nel vostro programma Python, questo significa che l'intero programma crollerà. È una cosa che non si vuole succeda nei programmi reali: si vuole invece che il programma individui gli errori, li gestisca e poi continui la sua esecuzione. Per esempio, considerate il programma seguente, che contiene un errore di “divisione per zero”. Aprite una nuova finestra di file editor, inserite il codice seguente e salvatelo come *zeroDivide.py*.

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Abbiamo definito una funzione *spam*, le abbiamo dato un parametro e poi abbiamo stampato il valore di quella funzione con vari parametri per vedere che cosa succede. Questo è l'output che otterrete eseguendo il codice precedente:

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

Un errore *ZeroDivisionError* si verifica quando si tenta di dividere un numero per zero. Dal numero di riga indicato, si sa che l'errore è causato dall'enunciato *return* in *spam()*.

Gli errori possono essere gestiti con enunciati *try* ed *except*. Il codice che potrebbe contenere un errore è inserito in una clausola *try*. L'esecuzione del programma passa all'inizio di una successiva clausola *except*, se si verifica un errore. Si può inserire il precedente codice che contiene una divisione per zero in una clausola *try* e in una clausola *except* si inserisce il codice per gestire quello che succede quando si verifica l'errore.

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Errore: Argomento non valido.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Quando il codice in una clausola *try* provoca un errore, l'esecuzione del programma passa

immediatamente al codice contenuto nella clausola except. Eseguito tale codice, l'esecuzione continua normalmente. L'output del programma precedente è questo:

```
21.0  
3.5  
Errore: Argomento non valido.  
None  
42.0
```

Notate che verrà identificato qualsiasi errore che si verifica in chiamate di funzione in un blocco try. Prendete il programma seguente, che invece ha le chiamate spam() nel blocco try:

```
def spam(divideBy):  
    return 42 / divideBy  
  
try:  
    print(spam(2))  
    print(spam(12))  
    print(spam(0))  
    print(spam(1))  
except ZeroDivisionError:  
    print('Errore: Argomento non valido.')
```

Quando questo programma viene eseguito, l'output è come questo:

```
21.0  
3.5  
Argomento non valido.
```

Il motivo per cui print(spam(1)) non viene mai eseguito è perché quando l'esecuzione salta al codice nella clausola except non torna all'inizio della clausola, ma continua come normalmente.

## Un breve programma: Indovina il numero

Gli esempi visti fin qui sono utili per introdurre concetti di base, ma vediamo ora come tutto quello che avete imparato può essere combinato nella scrittura di un programma più completo.

In questa sezione, vi presenterò un semplice gioco “Indovina il numero”. Quando eseguirete il programma, l'output sarà qualcosa di simile a questo:

Sto pensando a un numero fra 1 e 20.

Prova a indovinare.

**10**

Troppo basso.

Prova a indovinare.

**15**

Troppo basso.

Prova a indovinare.

**17**

Troppo alto.

Prova a indovinare.

**16**

Bene! Hai indovinato il numero in 4 passi!

Scrivete nel file editor il codice seguente e salvate il file come *guessTheNumber.py*:

```
# Questo è un gioco in cui si deve indovinare un numero.
import random
secretNumber = random.randint(1, 20)
print('Sto pensando a un numero fra 1 e 20.')

# Chiede al giocatore per sei volte di indovinare.
for guessesTaken in range(1, 7):
    print('Prova a indovinare.')
    guess = int(input())

    if guess < secretNumber:
        print('Troppo basso.')
    elif guess > secretNumber:
        print('Troppo alto.')
    else:
        break      # Questa condizione è il numero giusto!

if guess == secretNumber:
    print('Bene! Hai indovinato il numero in ' + str(guessesTaken) + ' passi!')
else:
    print('Niente. Il numero che avevo pensato è ' + str(secretNumber))
```

Esaminiamo il codice riga per riga, partendo dall'inizio.

```
# Questo è un gioco in cui si deve indovinare un numero.
import random
secretNumber = random.randint(1, 20)
```

Un commento all'inizio del codice spiega che cosa fa il programma. Poi, il programma importa il modulo `random` per poter usare la funzione `random.randint()` e generare un numero che l'utente dovrà indovinare. Il valore restituito, un intero casuale fra 1 e 20, è memorizzato nella variabile `secretNumber`.

```
print('Sto pensando a un numero fra 1 e 20.')
```

```
# Chiede al giocatore per sei volte di indovinare.
for guessesTaken in range(1, 7):
    print('Prova a indovinare.')
    guess = int(input())
```

Il programma dice al giocatore che ha pensato un numero segreto e concederà al giocatore sei possibilità per indovinarlo. Il codice che permette al giocatore di inserire la sua congettura e che controlla il numero inserito è in un ciclo `for` che verrà eseguito al massimo sei volte. La prima cosa che accade nel ciclo è che il giocatore inserisce un numero. Poiché `input()` restituisce una stringa, il valore restituito viene passato direttamente a `int()`, che converte la stringa in un intero. Questo viene memorizzato nella variabile `guess`.

```
if guess < secretNumber:
    print('Troppo basso.')
elif guess > secretNumber:
    print('Troppo alto.)
```

Queste righe di codice verificano se il valore ipotizzato dal giocatore è minore o maggiore del numero segreto.

In ciascun caso, viene stampato sullo schermo un indizio.

```
else:  
    break # Questa condizione è il numero giusto!
```

Se il valore inserito non è né più alto né più basso del numero segreto, deve essere uguale al numero segreto, nel qual caso l'esecuzione del programma esce dal ciclo for.

```
if guess == secretNumber:  
    print('Bene! Hai indovinato il numero in ' + str(guessesTaken) + ' guesses!')  
else:  
    print('Niente. Il numero che avevo pensato è ' + str(secretNumber))
```

Dopo il ciclo for, un enunciato if...else verifica se il giocatore ha indovinato il numero e stampa sullo schermo un messaggio opportuno. In entrambi i casi, il programma visualizza una variabile che contiene un valore intero (guessesTaken e secretNumber). Dato che deve concatenare questi valori interi a stringhe, passa queste variabili alla funzione str(), che restituisce la forma stringa di quegli interi. Ora queste stringhe possono essere concatenate con gli operatori +, per poi essere infine passate alla chiamata della funzione print().

## Riepilogo

Le **funzioni** sono il modo principale per suddividere il codice in compartimenti logicamente definiti. Dato che le **variabili** nelle funzioni esistono solo nel loro **ambito locale**, il codice in una funzione non può influenzare direttamente i valori delle variabili in altre funzioni. Questo pone dei limiti a quali parti del codice possano modificare i valori delle variabili, il che può essere molto utile quando si deve effettuare il **debug del codice**.

Le funzioni sono uno strumento eccellente per **organizzare il codice**. Potete pensarle come **scatole nere**: hanno input sotto forma di **parametri** e output sotto forma di **valori restituiti** e il codice al loro interno non influenza le variabili di altre funzioni.

Nei capitoli precedenti, un singolo errore è sufficiente per mandare in tilt i programmi. In questo capitolo, avete visto gli enunciati try ed except, che possono eseguire del codice quando viene rilevato un errore. Questo può rendere i vostri programmi più resistenti ai casi di errore più comuni.

## Domande di ripasso

1. Perché è vantaggioso avere le funzioni nei programmi?
2. Quando viene eseguito il codice in una funzione: quando la funzione è definita o quando viene chiamata?
3. Quale enunciato crea una funzione?
4. Qual è la differenza fra una funzione e una chiamata di funzione?
5. Quanti ambiti globali vi sono in un programma Python? Quanti ambiti locali?
6. Che cosa succede alle variabili in un ambito locale quando la chiamata di funzione ritorna?
7. Che cos'è un valore di ritorno? Un valore di ritorno può far parte di un'espressione?
8. Se una funzione non ha un enunciato return, qual è il valore restituito da una chiamata a quella

funzione?

9. Come si può forzare una variabile in una funzione perché faccia riferimento alla variabile globale?
10. Qual è il tipo di dati di `None`?
11. Che cosa fa l'enunciato `import tuttiituoanimalisichiamanoeric`?
12. Se avessi una funzione `bacon()` in un modulo che si chiama `spam`, come la chiameresti dopo aver importato `spam`?
13. Come si può impedire che un programma vada in crash quando incontra un errore?
14. Che cosa entra nella clausola `try`? Che cosa entra nella clausola `except`?

## Un po' di pratica

Per esercitarvi, scrivete dei programmi che svolgono le attività seguenti.

### La successione di Collatz

Scrivete una funzione `collatz()` che ha un parametro `number`. Se `number` è pari, allora `collatz()` deve stampare `number // 2` e restituisce questo valore. Se `number` è dispari, `collatz()` deve stampare e restituire `3 * number + 1`. Quindi scrivete un programma che consenta all'utente di scrivere un intero e che continui a chiamare `collatz()` per quel numero fino a che la funzione restituisce il valore 1. (Abbastanza curiosamente, questa successione funziona effettivamente per qualsiasi intero – prima o poi, sviluppando questa successione, si arriva sempre a 1. Anche i matematici non sanno bene il perché.) Il vostro programma esplora quella che è chiamata la *successione di Collatz*, a volte definita “il più semplice fra i problemi matematici impossibili”.

Ricordatevi di convertire il valore restituito da `input()` in un intero con la funzione `int()`; altrimenti, sarà un valore stringa.

Suggerimento: un `number` intero è pari se `number % 2 == 0`, è dispari se `number % 2 == 1`.

L'output di questo programma può essere di questo tipo:

Inserisci un numero:

```
3  
10  
5  
16  
8  
4  
2  
1
```

### Convalida dell'input

Aggiungete enunciati `try` ed `except` al progetto precedente, per stabilire se l'utente inserisce qualcosa che non è un intero. Normalmente, la funzione `int()` provocherà un errore `ValueError` se le viene passata una stringa che non è un intero, come in `int('puppy')`. Nella clausola `except`, stampate un messaggio per l'utente in cui gli dite che deve inserire un numero intero.

# Liste

Un ulteriore argomento che dovete conoscere per poter iniziare a scrivere programmi sul serio è il **tipo di dati lista**, con la sua parente stretta, la **tupla**. Liste e tuple possono contenere più valori, il che rende più facile scrivere programmi per **gestire grandi quantità di dati**. Poiché le liste possono avere come elementi altre liste, si può utilizzarle per organizzare i dati in **strutture gerarchiche**.

In questo capitolo, analizzeremo gli aspetti fondamentali delle liste. Parleremo anche dei metodi, che sono funzioni collegate a valori di un certo tipo di dati. Poi esamineremo le tuple, che assomigliano alle liste, e i tipi di dati stringa e le loro differenze rispetto ai valori lista. Nel prossimo capitolo vedremo invece il tipo di dati dizionario.

## Il tipo di dati lista

Una **lista** è un valore che contiene più valori in una successione ordinata. Il termine *valore lista* si riferisce alla lista stessa (che è un valore che può essere memorizzato in una variabile o passato a una funzione come qualsiasi altro valore), non ai valori all'interno del valore lista. Un valore lista ha un aspetto come questo: `['gatto', 'pipistrello', 'topo', 'elefante']`. Come i valori stringa vanno racchiusi fra apici per indicare dove la stringa inizia e termina, una lista inizia con una parentesi quadra aperta e termina con una parentesi quadra chiusa, `[]`. I valori all'interno della lista sono chiamati **elementi della lista** (o, con termine inglese, **item**). Gli elementi sono separati da virgolette (sono, cioè, *delimitati da virgole*, in inglese *comma-delimited*). Per esempio, inserite quanto segue nella shell interattiva:

```

>>> [1, 2, 3]
[1, 2, 3]
>>> ['gatto', 'pipistrello', 'topo', 'elefante']
['gatto', 'pipistrello', 'topo', 'elefante']
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['gatto', 'pipistrello', 'topo', 'elefante']
>>> spam
['gatto', 'pipistrello', 'topo', 'elefante']

```

Alla variabile `spam` ❶ viene assegnato ancora un solo valore: il valore lista a sua volta contiene altri valori. Il valore `[]` è una lista vuota che non contiene valori, analoga a `" "`, la stringa vuota.

## Ottenere i singoli valori in una lista mediante gli indici

Supponiamo di avere la lista `['gatto', 'pipistrello', 'topo', 'elefante']` memorizzata in una variabile `spam`. Il codice Python `spam[0]` avrebbe come valore `'gatto'`, `spam[1]` avrebbe come valore `'pipistrello'` e così via. L'intero fra parentesi quadre è chiamato **indice**. Il primo valore della lista ha indice 0, il secondo indice 1, il terzo indice 2 ecc.

La [Figura 4.1](#) mostra un valore lista assegnato a `spam`, e le espressioni indicizzate con i relativi valori.



**Figura 4.1** - Un valore lista memorizzato nella variabile `spam`, con l'indicazione dei valori a cui si riferiscono i vari indici.

Per esempio, scrivete nella shell interattiva queste espressioni. Iniziate assegnando una lista alla variabile `spam`.

```

>>> spam = ['gatto', 'pipistrello', 'topo', 'elefante']
>>> spam[0]
'gatto'
>>> spam[1]
'pipistrello'
>>> spam[2]
'topo'
>>> spam[3]
'elefante'
>>> ['gatto', 'pipistrello', 'topo', 'elefante'][3]
'elefante'
❶ >>> 'Ciao ' + spam[0]
'Ciao gatto'
❷ >>> 'Il ' + spam[1] + ' ha mangiato il ' + spam[0] + '.'
'Il pipistrello ha mangiato il gatto.'

```

Notate che l'espressione `'Ciao ' + spam[0]` ❶ viene valutata `'Ciao ' + 'gatto'` perché `spam[0]` viene valutata alla stringa `'gatto'`. Questa espressione a sua volta viene valutata dando il valore stringa `'Ciao gatto'` ❷. Python vi dà un messaggio d'errore `IndexError` se usate un indice superiore al numero dei valori nel valore lista.

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'elefante']
```

```
>>> spam[10000]
```

Traceback (most recent call last):

```
  File "<pyshell#9>", line 1, in <module>
```

```
    spam[10000]
```

IndexError: list index out of range

Gli indici possono essere solo valori interi, non in virgola mobile. L'esempio seguente provocherà un errore TypeError:

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'elefante']
```

```
>>> spam[1]
```

```
'pipistrello'
```

```
>>> spam[1.0]
```

Traceback (most recent call last):

```
  File "<pyshell#13>", line 1, in <module>
```

```
    spam[1.0]
```

TypeError: list indices must be integers, not float

```
>>> spam[int(1.0)]
```

```
'pipistrello'
```

Le liste possono contenere anche altri valori lista. Ai valori in queste liste di liste si può accedere utilizzando più indici, come in questo caso:

```
>>> spam = [['gatto', 'pipistrello'], [10, 20, 30, 40, 50]]
```

```
>>> spam[0]
```

```
['gatto', 'pipistrello']
```

```
>>> spam[0][1]
```

```
'pipistrello'
```

```
>>> spam[1][4]
```

```
50
```

Il primo indice dice quale valore lista usare, il secondo indica il valore all'interno di quel valore lista. Per esempio, spam[0][1] si riferisce a 'pipistrello', il secondo valore nella prima lista. Se usate un solo indice, il programma stampa tutta la lista che è il valore che ha quell'indice.

## Indici negativi

Gli indici normalmente partono da 0 e crescono, ma si possono usare anche **indici negativi**. Il valore intero -1 si riferisce all'ultimo indice in una lista, il valore -2 si riferisce al penultimo indice in una lista, e così via. Inserite nella shell interattiva quanto segue:

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'leone']
```

```
>>> spam[-1]
```

```
'leone'
```

```
>>> spam[-3]
```

```
'pipistrello'
```

```
>>> 'Il ' + spam[-1] + ' ha paura del ' + spam[-3] + '.'
```

```
'Il leone ha paura del pipistrello.'
```

## Sottoliste a sezioni

Come un indice può estrarre un singolo valore da una lista, una **slice (sezione)** può estrarre più valori da una lista, sotto forma di una nuova lista. Una sezione viene scritta fra **parentesi quadre**, come un indice, ma è formata da due indici separati da un segno di due punti. Notate la differenza fra indici e *sezioni*.

- `spam[2]` è una lista con un indice (un intero).
- `spam[1:4]` è una lista con una sezione (due interi).

In una sezione, il primo indice è l'indice da cui inizia la sezione. Il secondo intero è l'indice a cui la sezione termina. Una sezione arriva fino al valore del secondo indice, senza includerlo. Una sezione ha come valore un nuovo valore lista. Inserite quanto segue nella shell interattiva:

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'leone']
>>> spam[0:4]
['gatto', pipistrello', 'topo', 'leone']
>>> spam[1:3]
['pipistrello', 'topo']
>>> spam[0:-1]
['gatto', 'pipistrello', 'topo']
```

Come scorcianoia, potete omettere uno o entrambi gli indici ai due lati del segno di due punti nella sezione. Se si omette il primo indice è come se si usasse 0: si indica cioè l'inizio della lista. Se si omette il secondo indice è come usare la lunghezza della lista, quindi la slice terminerà alla fine della lista. Inserite quanto segue nella shell interattiva:

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'leone']
>>> spam[:2]
['gatto', 'pipistrello']
>>> spam[1:]
['pipistrello', 'topo', 'leone']
>>> spam[:]
['gatto', 'pipistrello', 'topo', 'leone']
```

## Ottenere la lunghezza di una lista con `len()`

Se le si passa come argomento un valore lista, la funzione `len()` restituisce il numero dei valori nella lista, così come conta il numero dei caratteri, se le viene passato come argomento un valore stringa. Inserite quanto segue nella shell interattiva:

```
>>> spam = ['gatto', 'cane', 'alce']
>>> len(spam)
3
```

## Modificare i valori in una lista con gli indici

Normalmente il nome di una variabile si trova sul lato sinistro di un enunciato di assegnazione, come in `spam = 42`. Tuttavia, si può anche usare l'indice di una lista per modificare il valore che si trova a quell'indice. Per esempio, `spam[1] = 'oritteropo'` significa "Assegna all'elemento che si trova all'indice 1

nella lista spam il valore stringa 'oritteropo'. Provate a inserire quanto segue nella shell interattiva:

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'elefante']
>>> spam[1] = 'oritteropo'
>>> spam
['gatto', 'oritteropo', 'topo', 'elefante']
>>> spam[2] = spam[1]
>>> spam
['gatto', 'oritteropo', 'oritteropo', 'elefante']
>>> spam[-1] = 12345
>>> spam
['gatto', 'oritteropo', 'oritteropo', 12345]
```

## Concatenazione e replica di liste

L'operatore + può **concatenare due liste** creando un nuovo valore lista esattamente come può concatenare due stringhe dando un nuovo valore stringa. Si può usare anche l'operatore \* con una lista e un valore intero per **replicare la lista**. Inserite quanto segue nella shell interattiva:

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

## Eliminare valori dalle liste con enunciati del

Gli enunciati del eliminano il valore che si trova a un determinato indice in una lista. L'indice di tutti i valori nella lista successivi al valore eliminato diminuirà di una unità. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'elefante']
>>> del spam[2]
>>> spam
['gatto', 'pipistrello', 'elefante']
>>> del spam[2]
>>> spam
['gatto', 'pipistrello']
```

L'enunciato del può essere usato anche su una variabile semplice per eliminarla, come se fosse una sorta di enunciato di “disassegnazione”. Se provate a usare la variabile dopo averla eliminata, otterrete un errore NameError, perché la variabile non esiste più.

Nella pratica, non succede quasi mai di aver bisogno di eliminare variabili semplici. L'enunciato del viene utilizzato in prevalenza per eliminare valori dalle liste.

## Lavorare con le liste

Quando si inizia a scrivere programmi, si è tentati di creare molte variabili individuali per

memorizzare un gruppo di valori simili. Per esempio, se volessi memorizzare i nomi dei miei gatti, potrei essere tentato di scrivere codice come questo:

```
catName1 = 'Zophie'  
catName2 = 'Pooka'  
catName3 = 'Simon'  
catName4 = 'Lady Macbeth'  
catName5 = 'Fat-tail'  
catName6 = 'Miss Cleo'
```

Questo è un pessimo modo di scrivere codice. Da un lato, se il numero dei gatti cambia, il programma non sarà mai in grado di memorizzare più gatti di quante sono le variabili esistenti. Questi tipi di programmi hanno anche molto codice duplicato o pressoché identico. Considerate quanto codice viene duplicato nel programma che segue, che dovreste inserire nel file editor e salvare con il nome *allMyCats1.py*:

```
print('Inserisci il nome del gatto 1:')  
catName1 = input()  
print('Inserisci il nome del gatto 2:')  
catName2 = input()  
print('Inserisci il nome del gatto 3:')  
catName3 = input()  
print('Inserisci il nome del gatto 4:')  
catName4 = input()  
print('Inserisci il nome del gatto 5:')  
catName5 = input()  
print('Inserisci il nome del gatto 6:')  
catName6 = input()  
print('I nomi dei gatti sono:')  
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +  
catName5 + ' ' + catName6)
```

Anziché usare molte variabili ripetitive, si può usare un'unica variabile che contiene un valore lista. Per esempio, ecco una nuova versione migliorata del programma *allMyCats1.py*. Questa nuova versione usa un'unica lista e può memorizzare qualsiasi numero di gatti di cui l'utente scriva il nome. In una nuova finestra del file editor, scrivete il codice sorgente che segue e salvatelo con il nome *allMyCats2.py*:

```
catNames = []  
while True:  
    print('Inserisci il nome del gatto ' + str(len(catNames) + 1) +  
          ' (Oppure non scrivere niente per finire.)')  
    name = input()  
    if name == '':  
        break  
    catNames = catNames + [name] # list concatenation  
print('I nomi dei gatti sono:')  
for name in catNames:  
    print(' ' + name)
```

Se eseguite il programma, l'output sarà simile a questo:

```
Inserisci il nome del gatto 1 (Oppure non scrivere niente per finire.):
```

**Zophie**

Inserisci il nome del gatto 2 (Oppure non scrivere niente per finire.):

**Pooka**

Inserisci il nome del gatto 3 (Oppure non scrivere niente per finire.):

**Simon**

Inserisci il nome del gatto 4 (Oppure non scrivere niente per finire.):

**Lady Macbeth**

Inserisci il nome del gatto 5 (Oppure non scrivere niente per finire.):

**Fat-tail**

Inserisci il nome del gatto 6 (Oppure non scrivere niente per finire.):

**Miss Cleo**

Inserisci il nome del gatto 7 (Oppure non scrivere niente per finire.):

I nomi dei gatti sono:

Zophie

Pooka

Simon

Lady Macbeth

Fat-tail

Miss Cleo

Il vantaggio di usare una lista è che i dati ora sono in una struttura, perciò il vostro programma possiede una flessibilità nell'elaborazione dei dati molto maggiore di quella che avrebbe potuto avere con molte variabili ripetitive.

## Uso di cicli for con le liste

Nel [Capitolo 2](#), abbiamo visto come usare i cicli for per eseguire un blocco di codice un certo numero di volte. Tecnicamente, un ciclo for ripete il blocco di codice una volta per ciascun valore in un valore lista (o simile a una lista). Per esempio, se eseguite:

```
for i in range(4):
    print(i)
```

l'output del programma sarà:

```
0
1
2
3
```

Questo perché il valore restituito da range(4) è un valore simile a una lista che Python considera analogo a [0, 1, 2, 3] Il programma seguente ha lo stesso output del programma precedente:

```
for i in [0, 1, 2, 3]:
    print(i)
```

Quello che fa il precedente ciclo for in realtà è ripetere la sua clausola con la variabile i impostata ai successivi valori nella lista [0, 1, 2, 3] per ciascuna iterazione.

**NOTA**

In questo libro uso l'espressione "simile a una lista" per indicare tipi di dati che tecnicamente sono chiamati *successioni*. Non c'è bisogno che conosciate la definizione tecnica di questo termine.

Una tecnica comune in Python è quella di usare `range(len(unaLista))` con un ciclo `for` per iterare sugli indici di una lista. Per esempio, inserite quanto segue nella shell:

```
>>> scorte = ['penne', 'cucitrici', 'lanciafiamme', 'raccoglitori']
>>> for i in range(len(scorte)):
    print('Indice ' + str(i) + ' in scorte è: ' + scorte[i])
Indice 0 in scorte è: penne
Indice 1 in scorte è: cucitrici
Indice 2 in scorte è: lanciafiamme
Indice 3 in scorte è: raccoglitori
```

L'uso di `range(len(scorte))` in questo ciclo `for` è comodo perché il codice nel ciclo può accedere all'indice (come variabile `i`) e al valore che si trova a quell'indice (come `scorte[i]`). L'aspetto più interessante e più utile è che `range(len(scorte))` itererà su tutti gli indici di `scorte`, non importa quanti siano gli elementi che contiene.

## Gli operatori `in` e `not in`

Si può stabilire se un certo valore è presente o meno in una lista mediante gli operatori `in` e `not in`. Come altri operatori, anche questi sono utilizzati nelle espressioni e connettono due valori: il valore che va cercato nella lista e la lista in cui si vuol sapere se è presente o meno. Queste espressioni avranno un valore Booleano. Inserite quanto segue nella shell interattiva:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

Per esempio, il programma seguente consente all'utente di scrivere il nome di un animale e poi controlla se il nome è presente in una lista di animali domestici. Aprite una nuova finestra di file editor, inserite il codice seguente e salvatelo con il nome `myPets.py`:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Inserisci il nome di un animale domestico')
name = input()
if name not in myPets:
    print('Non ho un animale domestico di nome ' + name)
else:
    print(name + ' è uno dei miei animali domestici')
```

L'output sarà qualcosa di simile a questo:

Inserisci il nome di un animale domestico:

**Footfoot**

Non ho un animale domestico di nome Footfoot

## Il trucco dell'assegnazione multipla

Il trucco dell'**assegnazione multipla** è una scorciatoia che permette di assegnare a più variabili i valori contenuti in una lista, con una sola riga di codice. Così, invece di:

```
>>> gatto = ['grasso', 'nero', 'chiacchierone']
>>> dimensione = gatto[0]
>>> colore = gatto[1]
>>> carattere = gatto[2]
```

si può scrivere questa riga di codice:

```
>>> gatto = ['grasso', 'nero', 'chiacchierone']
>>> dimensione, colore, carattere = gatto
```

Il numero delle variabili e la lunghezza della lista devono essere uguali, altrimenti Python dichiarerà un `ValueError`:

```
>>> gatto = ['grasso', 'nero', 'chiacchierone']
>>> dimensione, colore, carattere, nome = gatto
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    dimensione, colore, carattere, nome = gatto
ValueError: need more than 3 values to unpack
```

## Operatore di assegnazione aumentata

Quando si assegna un valore a una variabile, spesso si usa la variabile stessa. Per esempio, dopo aver assegnato il valore `42` alla variabile `spam`, si incrementa di 1 il valore in `spam` con questo codice:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

Come scorciatoia, si può usare l'operatore di assegnazione aumentata `+=`:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

Vi sono operatori di assegnazione aumentata per gli operatori `+`, `-`, `*`, `/` e `%`: sono descritti nella [Tabella 4.1](#).

**Tabella 4.1** - Gli operatori di assegnazione aumentata.

Enunciato di assegnazione aumentata	Enunciato di assegnazione equivalente
spam += 1	spam = spam + 1
spam -= 1	spam = spam - 1
spam *= 1	spam = spam * 1
spam /= 1	spam = spam / 1
spam %= 1	spam = spam % 1

L'operatore `+=` può anche effettuare la concatenazione di stringhe e liste e l'operatore `*=` può effettuare la replicazione di stringhe e liste. Inserite nella shell interattiva:

```
>>> spam = 'Ciao'
>>> spam += ' mondo!'
>>> spam
'Ciao mondo!'
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

## Metodi

Un **metodo** è come una funzione, tranne che viene **“chiamato su” un valore**. Per esempio, supponiamo che in `spam` sia memorizzato un valore lista; si potrebbe allora chiamare il metodo di lista `index()` (che spiegherò fra breve) in questo modo: `spam.index('ciao')`. Il metodo viene scritto dopo il valore, separato da un punto. Ciascun tipo di dati ha il proprio insieme di metodi. Il tipo di dati lista, per esempio, ha parecchi metodi utili per trovare, aggiungere, eliminare e manipolare variamente i valori contenuti in una lista.

## Trovare un valore in una lista con il metodo `index()`

I valori lista hanno un metodo `index()` a cui può essere passato un valore; se quel valore è presente nella lista, viene restituito l'indice di quel valore. Se il valore invece non è nella lista, Python emette un errore `ValueError`. Inserite quanto segue nella shell:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

Quando nella lista esistono duplicati del valore, viene restituito l'indice della sua prima occorrenza. Inserite quanto segue nella shell e notate che `index()` restituisce 1, non 3:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

## Aggiungere valori a liste con i metodi append() e insert()

Per **aggiungere nuovi valori a una lista**, si usano i metodi `append()` e `insert()`. Inserite quanto segue nella shell interattiva per chiamare il metodo `append()` su un valore lista memorizzato nella variabile `spam`:

```
>>> spam = ['gatto', 'cane', 'pipistrello']
>>> spam.append('alce')
>>> spam
['gatto', 'cane', 'pipistrello', 'alce']
```

La chiamata al metodo `append()` aggiunge l'elemento alla fine della lista (si dice anche che lo “accoda” alla lista). Il metodo `insert()` invece può inserire un valore in corrispondenza di qualsiasi indice della lista. Il primo argomento di `insert()` è l'indice per il nuovo valore, il secondo argomento è il nuovo valore da inserire. Inserite quanto segue nella shell interattiva:

```
>>> spam = ['gatto', 'cane', 'pipistrello']
>>> spam.insert(1, 'pollo')
>>> spam
['gatto', 'pollo', 'cane', 'pipistrello']
```

Noteate che il codice è `spam.append('alce')` e `spam.insert(1, 'pollo')`, non `spam = spam.append('alce')` e `spam = spam.insert(1, 'pollo')`. Né `append()` né `insert()` danno il nuovo valore di `spam` come proprio valore di ritorno. (In effetti, il valore di ritorno di `append()` e `insert()` è `None`, perciò decisamente non è il caso di memorizzarlo come nuovo valore della variabile.) La lista viene invece modificata “sul posto”. Della modifica della lista sul posto parleremo più dettagliatamente più avanti, in “[Tipi di dati modificabili e immutabili](#)” a pagina 86.

I metodi appartengono a un singolo tipo di dati. I metodi `append()` e `insert()` sono metodi di lista e possono essere chiamati solo per valori lista, non per altri valori come stringhe o interi. Inserite nella shell interattiva quanto segue, e notate i messaggi di errore `AttributeError` che si presentano:

```
>>> eggs = 'ciao'
>>> eggs.append('mondo')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    eggs.append('mondo')
AttributeError: 'str' object has no attribute 'append'
>>> bacon = 42
>>> bacon.insert(1, 'mondo')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    bacon.insert(1, 'mondo')
AttributeError: 'int' object has no attribute 'insert'
```

## Eliminare valori da liste con remove()

Al metodo `remove()` viene passato il valore che deve essere **eliminato dalla lista** su cui è chiamato. Inserite quanto segue nella shell interattiva:

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'elefante']
>>> spam.remove('pipistrello')
>>> spam
['gatto', 'topo', 'elefante']
```

Se si cerca di eliminare un valore che non è presente nella lista, si provoca un errore `ValueError`. Per esempio, inserite quanto segue nella shell interattiva e notate quale errore viene visualizzato:

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'elefante']
>>> spam.remove('pollo')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    spam.remove('pollo')
ValueError: list.remove(x): x not in list
```

Se il valore compare più volte nella lista, viene eliminata solo la sua prima occorrenza. Inserite quanto segue nella shell interattiva:

```
>>> spam = ['gatto', 'pipistrello', 'topo', 'gatto', 'cane', 'gatto']
>>> spam.remove('gatto')
>>> spam
['pipistrello', 'topo', 'gatto', 'cane', 'gatto']
```

L'*enunciato* del è comodo da usare quando si sa quale sia l'indice del valore che si vuole eliminare dalla lista. Il metodo `remove()` invece è adatto quando si sa quale sia il valore che si vuole eliminare, ma non necessariamente il posto che occupa nella lista.

## Ordinamento dei valori in una lista con il metodo `sort()`

Liste di valori numerici o liste di stringhe possono essere **ordinate** con il metodo `sort()`. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['api', 'gatti', 'cani', 'bufali', 'elefanti']
>>> spam.sort()
>>> spam
['api', 'bufali', 'cani', 'elefanti', 'gatti']
```

Si può anche passare `True` come argomento per parola chiave `reverse` in modo che `sort()` ordini i valori in ordine discendente. Inserite quanto segue nella shell interattiva:

```
>>> spam.sort(reverse=True)
>>> spam
['gatti', 'elefanti', 'camo', 'bufali', 'api']
```

Vi sono tre cose da notare a proposito del metodo `sort()`. La prima è che ordina le liste sul posto: non cercate di catturare il valore di ritorno scrivendo codice del tipo `spam = spam.sort()`. In secondo luogo, non si possono ordinare liste che contengono sia valori numerici sia valori stringa, poiché Python non

sa come confrontare questi valori.

Scrivete quanto segue nella shell interattiva e notate l'errore `TypeError`:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    spam.sort()
TypeError: unorderable types: str() < int()
```

In terzo luogo, `sort()` usa l'**ordine ASCII** anziché quello alfabetico tradizionale. Questo significa che le maiuscole vengono prima delle minuscole. Così, la lettera *a* minuscola viene dopo la *Z* maiuscola. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

Se dovete ordinare i valori secondo il convenzionale ordine alfabetico, potete passare `str.lower` come argomento per parola chiave `key` nella chiamata del metodo `sort()`.

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

Questo fa sì che il metodo `sort()` tratti tutti gli elementi nella lista come se fossero scritti in tutte minuscole, senza in realtà modificare i valori nella lista.

## Un programma di esempio: la palla magica con una lista

Con l'uso delle liste, potete scrivere una versione molto più elegante del programma della palla magica del [Capitolo 3](#). Invece delle molte righe di enunciati `elif` quasi identici, potete creare una lista unica su cui far lavorare il codice. Aprite una nuova finestra di file editor e inserite il codice seguente. Salvatelo poi con il nome `magic8Ball2.py`.

```
import random

messaggi = ['È sicuro',
    'Proprio così',
    'Sì'.
    'Risposta incerta, riprova',
    'Chiedi di nuovo più tardi',
    'Concentrati e chiedi di nuovo',
    'La mia risposta è no',
    'La prospettiva non è molto buona',
    'Molto dubbioso']

print(messaggi[random.randint(0, len(messaggi) - 1)])
```

## Eccezioni alle regole sui rientri del codice in Python

Nella maggior parte dei casi, la quantità di rientro di una riga di codice dice a Python in quale blocco si trova. Esistono però alcune eccezioni a questa regola. Per esempio, le liste possono disporsi su più righe nel file del codice sorgente. Il rientro di queste righe non ha importanza; Python sa che, finché non vede la parentesi quadra di chiusura, la lista non è completa.

Per esempio, potete avere del codice con questo aspetto:

```
spam = ['mele',
        'arance',
                    'banane',
['gatti']
print(spam)
```

Ovviamente, nella pratica, quasi tutti rispettano il comportamento di Python perché le loro liste abbiano un aspetto elegante e leggibile, come nel caso della lista di messaggi per il programma della palla magica.

Potete anche suddividere una singola istruzione su più righe utilizzando il carattere \ di “continuazione di riga” a fine della riga che volete mandare a capo. Potete pensare che \ dica “Questa istruzione continua sulla riga successiva”. Il rientro della riga dopo un carattere di continuazione \ non è dotato di significativo. Per esempio, quello che segue è codice Python valido:

```
print('Quattrocento e settantacinque ' +
      'anni fa...')
```

Questi piccoli trucchi sono utili quando si vogliono riorganizzare righe molto lunghe di codice Python in modo che risultino più facilmente leggibili (o, come in questo libro, quando la larghezza di pagina pone dei limiti invalicabili al contenuto di una sola riga stampata).

Quando eseguite questo programma, vedrete che funziona come *magic8Ball.py* del capitolo precedente.

Notate l'espressione utilizzata come indice per la lista messaggi: random.randint(0, len(messaggi) - 1). Questo produce un numero casuale da usare come indice, indipendentemente dal numero degli elementi di messaggi. In altre parole, si ottiene un numero casuale compreso fra 0 e il valore di len(messaggi) - 1.

Il vantaggio di questa impostazione è che si possono facilmente aggiungere ed eliminare stringhe nella lista messaggi senza dover modificare altre righe di codice. Se in seguito aggirate il vostro codice, ci saranno meno righe che dovete modificare e meno possibilità che le modifiche introducano qualche errore.

## Tipi simili a liste: stringhe e tuple

Le liste non sono gli unici tipi di dati che rappresentano successioni ordinate di valori. Stringhe e liste, per esempio, sono in realtà simili, se si pensa una stringa come una “lista” di singoli caratteri di testo.

Molte delle cose che si possono fare con le liste si possono fare anche con le stringhe: indicizzarle, estrarne delle **sezioni (slice)**; usarle per cicli for, con len() e con gli operatori in e not in.

Per verificarlo, inserite nella shell interattiva il codice seguente:

```

>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
    print('*****' + i + '*****')

```

```

***** Z *****
***** o *****
***** p *****
***** h *****
***** i *****
***** e *****

```

## Tipi di dati modificabili e immutabili

Liste e stringhe però sono diverse per un aspetto importante. Un valore **lista** è un **tipo di dati modificabile**: vi si possono aggiungere valori, eliminare valori o modificare valori. Una **stringa** invece è **immutabile**. Non può essere modificata. Se si cerca di riassegnare un singolo carattere in una stringa si ottiene un errore `TypeError`, come si può vedere inserendo nella shell interattiva quanto segue:

```

>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment

```

Il modo corretto per modificare una stringa è usare sezioni e concatenazione per costruire una nuova stringa copiando parti della vecchia stringa. Provate a inserire quanto segue nella shell interattiva:

```

>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
>>> newName
'Zophie the cat'

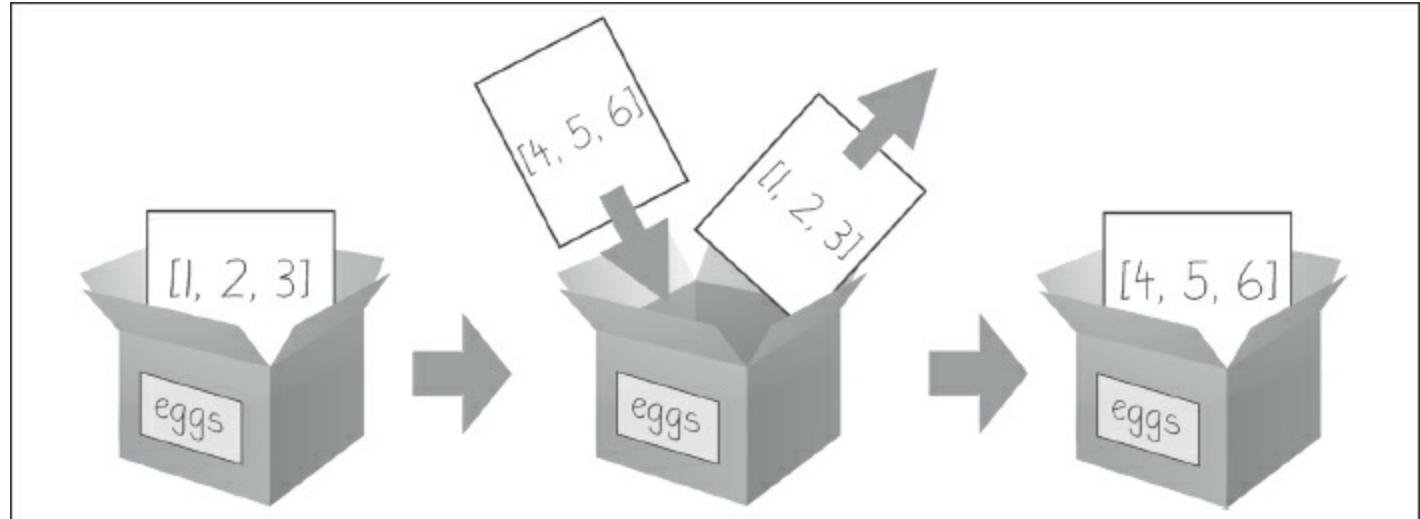
```

Abbiamo usato `[0:7]` e `[8:12]` per indicare i caratteri che non vogliamo sostituire. Notate che la stringa originale 'Zophie a cat' non viene modificata, perché le stringhe sono immutabili.

Anche se un valore lista è modificabile, la seconda riga nel codice seguente non modifica la lista `eggs`:

```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs
[4, 5, 6]
```

Il valore lista in `eggs` non viene modificato qui; viene invece creato un valore lista nuovo, diverso ([4, 5, 6]) che viene sovrascritto al vecchio valore lista ([1, 2, 3]). La situazione è rappresentata graficamente nella [Figura 4.2](#).

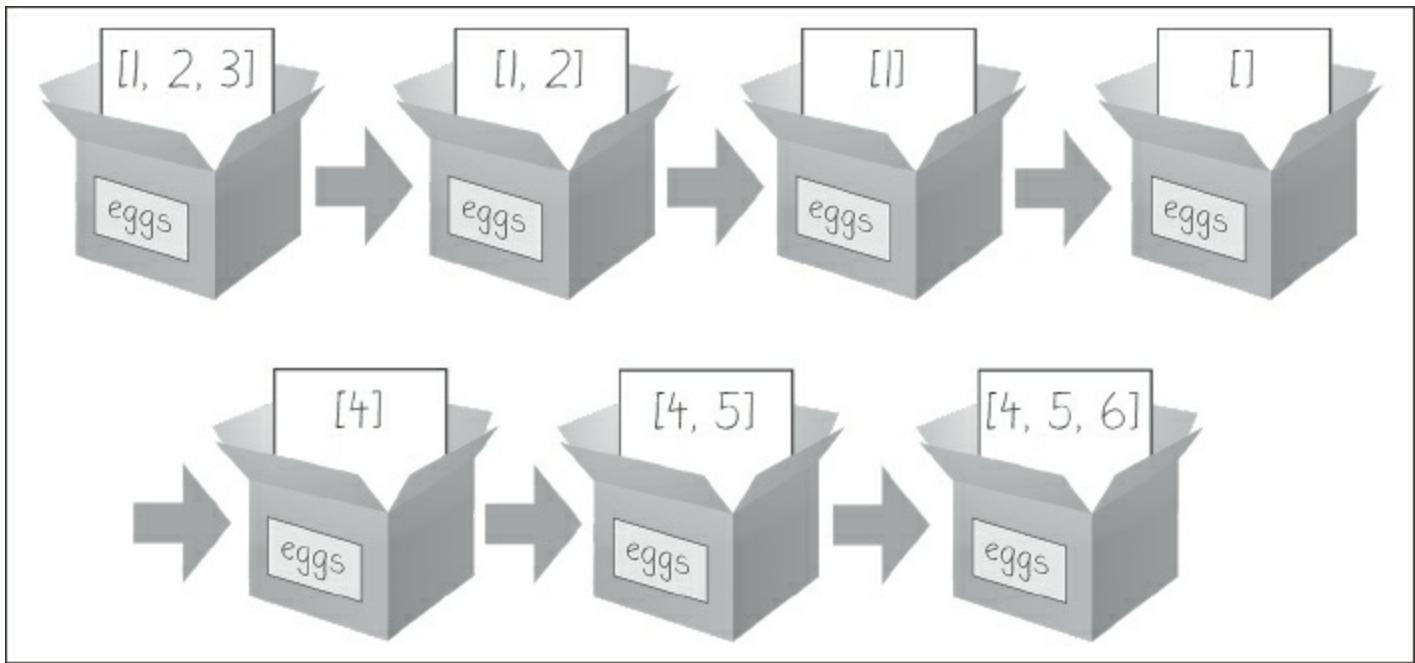


**Figura 4.2** - Quando viene eseguito `eggs = [4, 5, 6]`, i contenuti di `eggs` vengono sostituiti da un nuovo valore lista.

Se volete modificare effettivamente la lista originale in `eggs` perché contenga [4, 5, 6], dovete fare qualcosa di questo genere:

```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```

Nel primo esempio, il valore lista che ha `eggs` alla fine è lo stesso che aveva all'inizio. Semplicemente questa lista è stata modificata, anziché sovrascritta. La [Figura 4.3](#) rappresenta graficamente le sette modifiche effettuate dalle prime sette righe del precedente esempio nella shell interattiva.



**Figura 4.3** - L'enunciato del e il metodo append() modificano lo stesso valore lista “sul posto”.

Se si modifica un valore di un tipo di dati modificabile (quello che fanno l'enunciato `del` e il metodo `append()` nell'esempio precedente), si modifica il valore “sul posto”, perché il valore della variabile non è sostituito da un nuovo valore lista.

La distinzione fra tipi di dati modificabili e immutabili può sembrare oziosa, ma la sezione “[Passaggio di riferimenti](#)” a pagina 92 spiegherà le differenze di comportamento che si verificano quando si chiamano funzioni con argomenti modificabili oppure immutabili. prima però dobbiamo parlare del tipo di dati tupla, che è una forma immutabile del tipo di dati lista.

## Il tipo di dati tupla

Il tipo di dati **tupla** è quasi identico al tipo di dati lista, tranne che per due aspetti. In primo luogo, le tuple si scrivono racchiuse fra **parentesi tonde**, ( e ), e non fra parentesi quadre, [ e ]. Per esempio, inserite quanto segue nella shell interattiva.

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

Il motivo principale per cui le tuple si distinguono dalle liste, però, è che le tuple, come le stringhe, **sono immutabili**. Non è possibile modificare, eliminare o accodare valori in una tupla. Inserite quanto segue nella shell interattiva e guardate il messaggio di errore `TypeError`:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File '<pyshell#5>', line 1, in <module>
    eggs[1] = 99
```

Se avete solo un valore nella vostra tupla, potete indicarlo inserendo una virgola finale dopo il valore fra parentesi. Altrimenti, Python penserà che avete semplicemente scritto un valore fra normali parentesi. La virgola fa sapere a Python che si tratta di un valore tupla. (A differenza di quel che accade in altri linguaggi di programmazione, in Python è ammesso avere una virgola finale dopo l'ultimo elemento in una lista o in una tupla.) Inserite le seguenti chiamate alla funzione `type()` nella shell interattiva, per vedere la differenza:

```
>>> type('hello',)
<class 'tuple'
>>> type('hello')
<class 'str'>
```

Potete usare le tuple per far capire a chiunque legga il vostro codice che non volete che i valori di quella successione mutino. Se avete bisogno di una successione ordinata di valori che non cambi mai, usate una tupla. Un secondo vantaggio dell'uso delle tuple anziché delle liste è che, essendo immutabili e quindi non variando mai i loro contenuti, Python può realizzare qualche ottimizzazione, il che renderà il codice che usa le tuple un po' più veloce di quello che usa le liste.

## Conversione di tipi con le funzioni `list()` e `tuple()`

Come `str(42)` restituisce '42', la rappresentazione come stringa dell'intero 42, le funzioni `list()` e `tuple()` restituiscono versioni lista e tupla, rispettivamente, dei valori che vengono passati loro. Inserite nella shell interattiva quanto segue e notate che il valore restituito è di un tipo di dati diverso da quello del valore passato:

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(['cat', 'dog', 5])
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

È comodo convertire una tupla in una lista quando si ha bisogno di una versione modificabile di un valore tupla.

## Riferimenti

Come abbiamo visto, le variabili memorizzano valori stringa e valori interi. Inserite nella shell interattiva quanto segue:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

Avete assegnato 42 alla variabile `spam`, e poi avete copiato il valore che si trovava in `spam` e l'avete assegnato alla variabile `cheese`. Quando poi avete modificato in 100 il valore in `spam`, questo non ha avuto conseguenze per il valore in `cheese`, perché `spam` e `cheese` sono variabili diverse che memorizzano valori diversi.

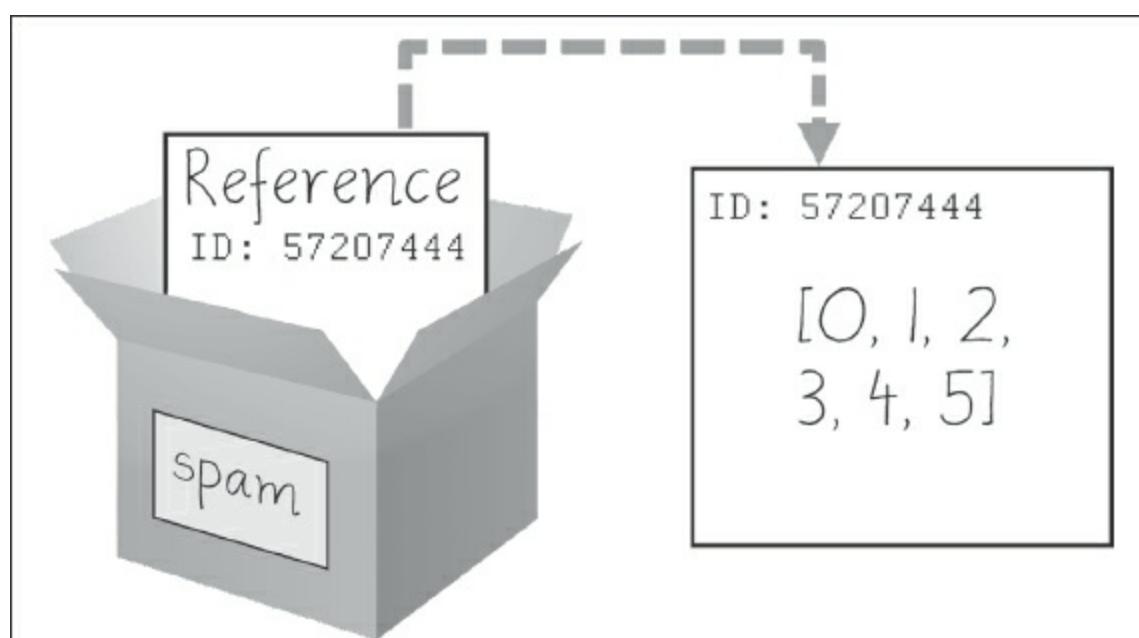
Le liste però non funzionano in questo modo. Quando si assegna una lista a una variabile, in realtà si assegna alla variabile il **riferimento a una lista**. Un **riferimento (reference)** è un valore che punta a qualche elemento di dati, e un riferimento a una lista è un valore che punta a una lista. Ecco del codice che renderà più facile comprendere questa distinzione. Inserite quanto segue nella shell interattiva:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

Potrà sembrarti strano. Il codice ha modificato solo la lista `cheese`, ma sembra che sia cambiata anche la lista `spam`.

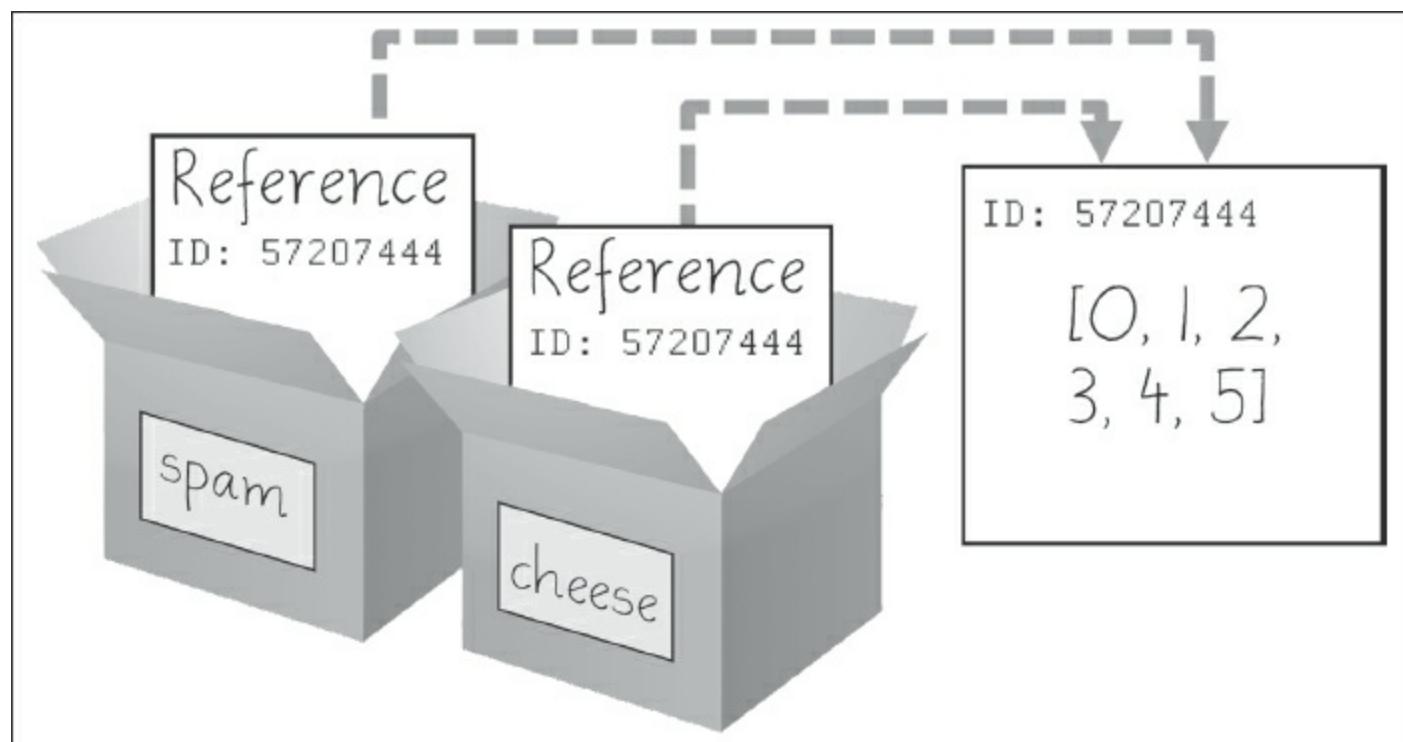
Quando si crea la lista ❶, si assegna un riferimento a quella lista nella variabile `spam`. Ma la riga successiva ❷ copia in `cheese` solo il riferimento alla lista che si trova in `spam` e non il valore lista stesso. Questo significa che i valori memorizzati in `spam` e `cheese` si riferiscono ora entrambi alla stessa lista. C'è solo una lista sottostante, perché la lista non viene mai copiata effettivamente. Così, quando si modifica il primo elemento di `cheese` ❸, si modifica la stessa lista a cui fa riferimento `spam`.

Ricordate che le variabili sono come scatole che contengono valori. Le figure precedenti in questo capitolo mostrano che le liste nelle scatole non sono proprio accurate, perché le variabili di lista non contengono effettivamente liste, ma solo riferimenti a liste. (Questi riferimenti avranno numeri identificativi, ID che Python usa internamente, ma che voi potrete tranquillamente ignorare.) Utilizzando la scatola come metafora per una variabile, la [Figura 4.4](#) mostra che cosa succede quando alla variabile `spam` viene assegnata una lista.

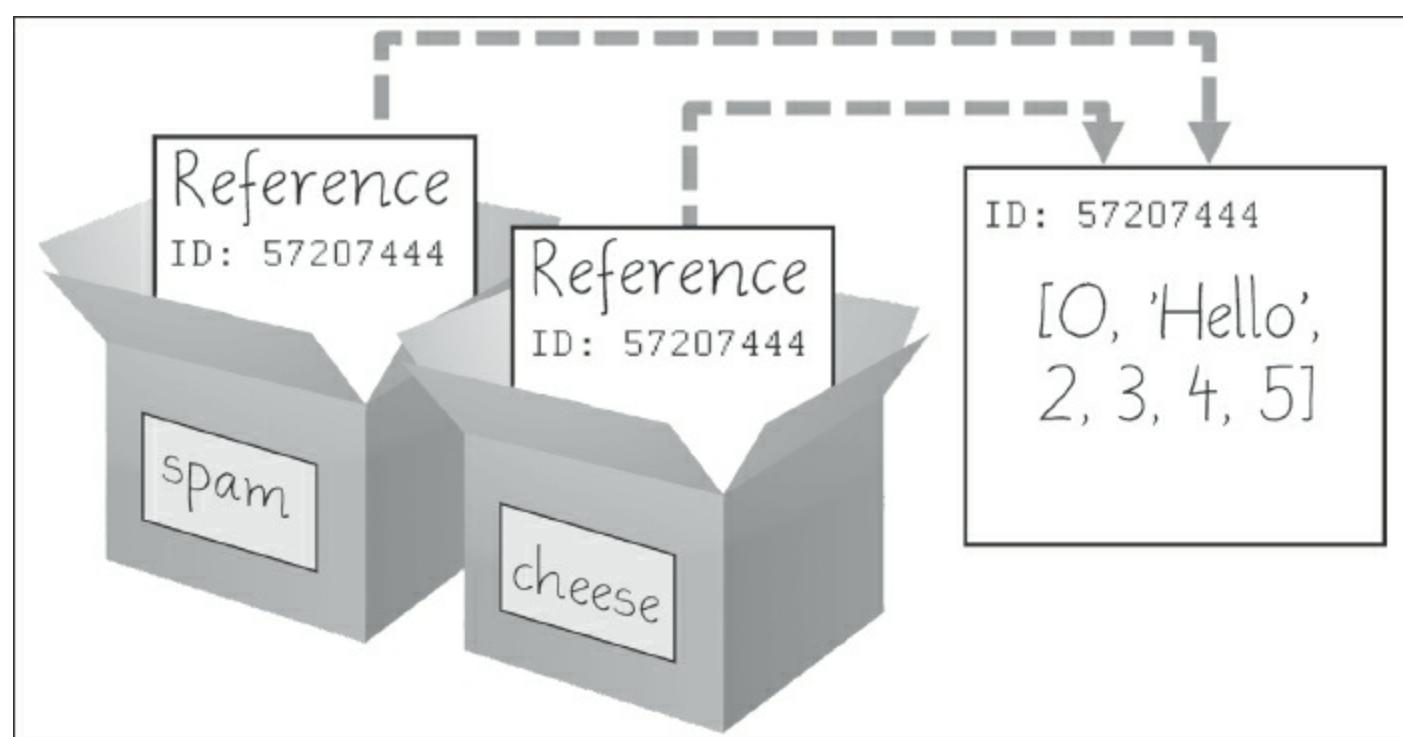


**Figura 4.4** - `spam = [0, 1, 2, 3, 4, 5]` memorizza un riferimento a una lista, non la lista effettiva.

Poi, nella [Figura 4.5](#), il riferimento in `spam` viene copiato in `cheese`. In `cheese` viene creato e memorizzato solo un nuovo riferimento, non una nuova lista. Notate come entrambi i riferimenti puntino alla stessa lista. Se si modifica la lista a cui si riferisce `cheese`, si modifica anche quella a cui fa riferimento `spam`, perché entrambe le variabili fanno riferimento alla stessa lista, come si può vedere nella [Figura 4.6](#).



**Figura 4.5** - `spam` = `cheese` copia il riferimento e non la lista.



**Figura 4.6** - `cheese[1] = 'Hello!'` modifica la lista a cui fanno riferimento entrambe le variabili.

Le variabili contengono riferimenti a valori lista e non i valori lista stessi, ma nel caso di valori stringa e interi le variabili contengono semplicemente il valore stringa o intero. Python usa riferimenti quando le variabili debbono memorizzare valori di tipi di dati modificabili, come le liste

o i dizionari. Per valori di tipi di dati immutabili, come stringhe, interi o tuple, le variabili di Python memorizzano i valori stessi.

Anche se le variabili in Python tecnicamente contengono riferimenti a valori lista o dizionario, spesso si dice per comodità (ma in modo impreciso) che la variabile contiene la lista o il dizionario.

## Passaggio di riferimenti

I riferimenti sono particolarmente importanti per capire come gli argomenti vengono passati alle funzioni. Quando viene chiamata una funzione, i valori degli argomenti vengono copiati nelle variabili parametro. Per le liste (e per i dizionari, di cui parleremo nel prossimo capitolo), questo significa che viene utilizzata per il parametro una copia del riferimento.

Per vedere che cosa ne consegue, aprite una nuova finestra di file editor, inserite il codice seguente e salvatelo con il nome *passingReference.py*:

```
def eggs(someParameter):
    someParameter.append('Hello')
spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notate che, quando viene chiamata `eggs()`, non viene utilizzato un valore di ritorno per assegnare un nuovo valore a `spam`: viene invece modificata direttamente sul posto la lista. Quando viene eseguito, il programma produce questo output:

```
[1, 2, 3, 'Hello']
```

Anche se `spam` e `someParameter` contengono riferimenti distinti, si riferiscono entrambe alla stessa lista. Per questo la chiamata del metodo `append('Hello')` all'interno della funzione influenza la lista, anche dopo il ritorno della chiamata di funzione.

Ricordate bene questo comportamento: se ci si dimentica che Python tratta in questo modo variabili di lista e dizionario si finisce per generare errori che fanno confusione.

## Le funzioni `copy()` e `deepcopy()` del modulo `copy`

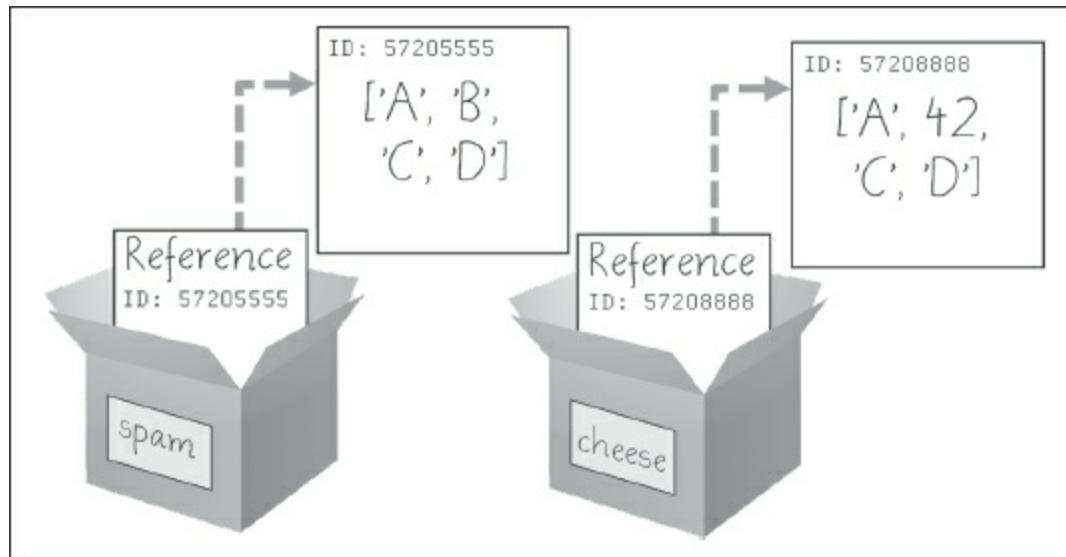
Anche se passare riferimenti è spesso il modo più comodo per trattare con liste e dizionari, se la funzione modifica la lista o il dizionario che le vengono passati è possibile che non si voglia che questi cambiamenti siano registrati nel valore lista o dizionario originale. Per questo motivo, Python mette a disposizione un modulo `copy` che offre due funzioni, `copy()` e `deepcopy()`. La prima, `copy.copy()`, può essere usata per effettuare **un duplicato di un valore modificabile**, come una lista o un dizionario, anziché per copiare solo un riferimento.

Inserite quanto segue nella shell interattiva:

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
```

[A', 42, 'C', 'D']

Ora le variabili `spam` e `cheese` si riferiscono a liste distinte, ed è questo il motivo per cui solo la lista in `cheese` viene modificata quando si assegna 42 all'indice 1. Come si può vedere nella [Figura 4.7](#), gli ID di riferimento non sono più identici per le due variabili, perché le variabili si riferiscono a liste indipendenti.



**Figura 4.7** - `cheese = copy.copy(spam)` crea una seconda lista che può essere modificata indipendentemente dalla prima.

Se la lista che dovete copiare contiene liste, usate la funzione `copy.deepcopy()` invece di `copy.copy()`. La funzione `deepcopy()` copierà anche queste liste interne.

## Riepilogo

Le **liste** sono tipi di dati utili perché consentono di scrivere codice che opera su un numero modificabile di valori in una singola variabile. Più avanti, vedrete programmi che usano liste per fare cose che sarebbe difficile o impossibile fare altrimenti.

Le **liste sono modificabili**: i loro contenuti, cioè, possono cambiare. **Tuple e stringhe**, anche se simili alle liste per qualche aspetto, **sono immutabili** e non possono essere modificate. Una variabile che contiene un valore tupla o stringa può essere sovrascritta con un nuovo valore tupla o stringa, ma non è la stessa cosa che modificare il valore esistente sul posto, come fanno, per esempio, i metodi `append()` o `remove()` sulle liste.

Le **variabili** non memorizzano direttamente valori lista; **memorizzano riferimenti** a liste. Questa è una distinzione importante, quando si copiano variabili o si passano liste come argomenti in chiamate di funzione. Poiché il valore che viene copiato è il riferimento alla lista, tenete presente che qualsiasi cambiamento apportato alla lista può avere conseguenze su altre variabili nel programma. Potete usare `copy()` o `deepcopy()` se volete modificare una lista in una variabile senza modificare la lista originaria.

## Domande di ripasso

1. Che cos'è `[]`?
2. Come assegnereste il valore ‘hello’ come terzo valore in una lista memorizzata in una

variabile che si chiama **spam**? (Supponete che **spam** contenga [2, 4, 6, 8, 10].

Per le tre domande seguenti, ipotizzate che **spam** contenga la lista ['a', 'b', 'c', 'd'].

3. Qual è il valore di `spam[int(int('3' * 2) / 11)]`?
4. Qual è il valore di `spam[-1]`?
5. Qual è il valore di `spam[:2]`?

Per le tre domande seguenti, ipotizzate che **bacon** contenga la lista [3.14, 'cat', 11, 'cat', True].

6. Qual è il valore di `bacon.index('cat')`?
7. Come risulta il valore lista in **bacon** dopo l'applicazione di `bacon.append(99)`?
8. Come risulta il valore lista in **bacon** dopo l'applicazione di `bacon.remove('cat')`?
9. Quali sono gli operatori di concatenazione e replicazione di liste?
10. Qual è la differenza tra i metodi di lista `append()` e `insert()`?
11. Indicate due modi per eliminare valori da una lista.
12. Indicate alcuni aspetti per cui i valori lista sono simili ai valori stringa.
13. Qual è la differenza fra liste e tuple?
14. Come si scrive il valore tupla che contiene solo il valore 42?
15. Come si può ottenere la forma tupla di un valore lista? Come si può ottenere la forma lista di un valore tupla?
16. Le variabili che "contengono" valori lista non contengono in effetti direttamente liste. Che cosa contengono invece?
17. Qual è la differenza fra `copy.copy()` e `copy.deepcopy()`?

## Un po' di pratica

Per esercitarvi, scrivete dei programmi che svolgano le attività seguenti.

### Virgole

Supponete di avere un valore lista come questo:

```
spam = ['mele', 'banane', 'tofu', 'gatti']
```

Scrivete una funzione che prenda un valore lista come argomento e restituisca una stringa con tutti gli elementi della lista separati da una virgola e uno spazio, con un *infine* inserito prima dell'ultimo elemento. Per esempio, passando la lista **spam** a questa funzione, il risultato sarebbe 'mele, banane, tofu, infine gatti'. La vostra funzione però deve poter funzionare con qualsiasi valore lista le sia passato.

### Immagini a caratteri

Supponiamo che abbiate una lista di liste, in cui ciascun valore nelle liste interne è una stringa di un carattere, come questa:

```
griglia = [['.', '.', '.', '.', '.', '.'],
           ['.', '0', '0', '.', '.', '.'],
           ['0', '0', '0', '0', '.', '.'],
           ['0', '0', '0', '0', '0', '.'],
           ['.', '0', '0', '0', '0', '0'],
           ['0', '0', '0', '0', '0', '.'],
           ['0', '0', '0', '0', '.', '.'],
           ['.', '0', '0', '.', '.', '.'],
           ['.', '.', '.', '.', '.', '.']]
```

Potete pensare `griglia[x][y]` come il carattere alle coordinate  $x$  e  $y$  di una “immagine” disegnata con caratteri di testo. L’origine  $(0, 0)$  sarà nell’angolo superiore sinistro, le coordinate  $x$  aumenteranno andando verso destra e le coordinate  $y$  aumenteranno procedendo dall’alto verso il basso.

Copiate il valore `griglia` precedente e scrivete del codice che la usi per stampare l’immagine.

```
..00.00..
.0000000.
.0000000.
..00000..
...000...
....0....
```

Suggerimento: dovrete usare un ciclo dentro un ciclo per stampare `griglia[0][0]`, poi `griglia[1][0]`, poi `griglia[2][0]` e così via, fino a `griglia[8][0]`; così avrete finito la prima riga, perciò stampate un ritorno a capo. Poi il vostro programma dovrà stampare `griglia[0][1]`, poi `griglia[1][1]`, poi `griglia[2][1]` e così via. L’ultima cosa che il vostro programma stamperà è `griglia[8][8]`.

Ricordatevi anche di passare a `print()` l’argomento per parola chiave `end`, se non volete che venga stampato automaticamente un a capo dopo ciascuna chiamata di `print()`.

# Dizionari e strutturazione dei dati

In questo capitolo, parleremo del tipo di dati **dizionario**, che offre un modo flessibile per **accedere ai dati** e **organizzarli**. Poi, combinando i dizionari con quello che già sapete delle liste, saprete come creare una **struttura di dati** per costruire il modello di una tavola da filetto.

## Il tipo di dati dizionario

Come una lista, un **dizionario** è una **collezione di molti valori**. A differenza degli indici delle liste, gli indici dei dizionari possono usare molti tipi di dati diversi, non solo interi. Gli indici dei dizionari sono chiamati **chiavi**, e una chiave con il relativo valore costituisce una **coppia chiave-valore**. Nel codice, un dizionario si racchiude fra parentesi graffe, {}. Inserite quanto segue nella shell interattiva:

```
>>> mioGatto = {'dimensioni': 'grasso', 'colore': 'grigio', 'carattere': 'allegro'}
```

Questo assegna alla variabile `mioGatto` un dizionario. Le chiavi del dizionario sono 'dimensioni', 'colore' e 'carattere'. I valori di queste chiavi sono 'grasso', 'grigio' e 'allegro', rispettivamente. Potete accedere a questi valori attraverso le loro chiavi:

```
>>> mioGatto['dimensioni']
'grasso'
>>> 'Il mio gatto ha la pelliccia di colore ' + mioGatto['colore'] + '.'
Il mio gatto ha la pelliccia di colore grigio.
```

I dizionari possono comunque usare valori interi come chiavi, come le liste usano gli interi per gli indici, ma non devono partire da 0 e possono utilizzare come chiavi dei numeri qualsiasi.

```
>>> spam = {12345: 'Combinazione lucchetto', 42: 'La risposta'}
```

## Dizionari e liste

A differenza di quel che avviene nelle liste, **gli elementi** che appartengono ai dizionari **non sono ordinati**. Il primo elemento in una lista di nome spam sarebbe spam[0]. Non esiste invece un “primo” elemento in un dizionario.

Mentre l’ordine degli elementi è importante per stabilire se due liste sono identiche, non importa in quale ordine le coppie chiave-valore vengano scritte all’interno di un dizionario.

Inserite quanto segue nella shell interattiva:

```
>>> spam = ['gatti', 'cani', 'alci']
>>> bacon = ['cani', 'alci', 'gatti']
>>> spam == bacon
False
>>> eggs = {'nome': 'Zophie', 'specie': 'gatto', 'età': '8'}
>>> ham = {'specie': 'gatto', 'età': '8', 'nome': 'Zophie'}
>>> eggs == ham
True
```

Poiché i dizionari non sono ordinati, non possono essere divisi in sezioni (*slice*) come le liste.

Se si tenta di accedere a una chiave che non esiste in un dizionario, si ottiene un messaggio d’errore KeyError, simile al messaggio d’errore IndexError di una lista. Inserite quanto segue nella shell interattiva, e notate il messaggio d’errore che viene presentato perché non esiste una chiave ‘colore’:

```
>>> spam = {'nome': 'Zophie', 'età': 7}
>>> spam['colore']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['colore']
KeyError: 'colore'
```

Anche se i dizionari non sono ordinati, il fatto che le chiavi possano avere valori arbitrari permette di **organizzare i dati** in modi potenti. Supponiamo che vogliate un programma in cui memorizzare i dati sui compleanni dei vostri amici. Potete usare un dizionario con i nomi come chiavi e le date di compleanno come valori.

Aprite una nuova finestra di file editor e inserite il codice seguente, poi salvatelo con il nome *birthdays.py*.

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dic 12', 'Carol': 'Mar 4'}
❷ while True:
❸     print('Inserisci un nome: (nulla per uscire)')
❹     name = input()
❺     if name == '':
❻         break

❾     if name in birthdays:
❿         print(birthdays[name] + ' è il compleanno di ' + name)
❽     else:
❾         print('Non ho informazioni sul compleanno di ' + name)
❿         print('In che giorno è il suo compleanno?')
❽         bday = input()
❾         birthdays[name] = bday
❽         print('Database dei compleanni aggiornato.')
```

Si crea un dizionario all'inizio e lo si memorizza in `birthdays` ❶. Si può vedere se il nome inserito esiste come chiave nel dizionario con la parola chiave in ❷, come si fa per gli elementi delle liste. Se il nome si trova nel dizionario, si può accedere al valore associato utilizzando le parentesi quadre ❸; in caso contrario, lo si può aggiungere utilizzando la medesima sintassi a parentesi quadre e l'operatore di assegnazione ❹.

Quando si esegue questo programma, si otterrà un output come questo:

Inserisci un nome: (nulla per uscire)

**Alice**

Apr 1 è il compleanno di Alice

Inserisci un nome: (nulla per uscire)

**Eve**

Non ho informazioni sul compleanno di Eve

In che giorno è il suo compleanno?

**Dic 5**

Database dei compleanni aggiornato.

Inserisci un nome: (nulla per uscire)

**Eve**

Dic 5 è il compleanno di Eve

Inserisci un nome: (nulla per uscire)

Ovviamente, tutti i dati che si inseriscono vengono dimenticati quando il programma termina. Imparerete come salvare i dati in file sul disco fisso nel [Capitolo 8](#).

## I metodi `keys()`, `values()` e `items()`

Esistono tre **metodi dei dizionari** che restituiscono valori, simili a liste, delle chiavi, dei valori o sia di chiavi che valori, di un dizionario: `keys()`, `values()` e `items()`. I valori restituiti da questi metodi non sono vere liste: non sono modificabili e non hanno un metodo `append()`, ma questi tipi di dati (`dict_keys`, `dict_values` e `dict_items`, rispettivamente) si *possono* utilizzare nei cicli `for`.

```
>>> spam = {'colore': 'rosso', 'età': 42}
>>> for v in spam.values():
    print(v)
rosso
42
```

Qui un ciclo `for` itera su ciascuno dei valori nel dizionario `spam`. Un ciclo `for` può iterare anche sulle chiavi o su chiavi e valori insieme:

```
>>> for k in spam.keys():
    print(k)
colore
età
>>> for i in spam.items():
    print(i)
('colore', 'rosso')
('età', 42)
```

Utilizzando i metodi `keys()`, `values()` e `items()`, un ciclo `for` può iterare su chiavi, valori o su coppie chiave-valore in un dizionario, rispettivamente. Notate che i valori nel valore `dict_items` restituito dal metodo

`items()` sono tuple di chiave e valore. Per ottenere una vera lista da uno di questi metodi, passate il suo valore di ritorno alla funzione `list()`:

```
>>> spam = {'colore': 'rosso', 'età': 42}
>>> spam.keys()
dict_keys(['colore', 'età'])
>>> list(spam.keys())
['colore', 'età']
```

La riga `list(spam.keys())` prende il valore `dict_keys` restituito da `keys()` e lo passa a `list()`, che poi restituisce un valore lista, `['colore', 'età']`.

È possibile utilizzare anche il trucco dell’assegnazione multipla in un ciclo `for` per assegnare chiave e valore a variabili distinte. Inserite quanto segue nella shell interattiva:

```
>>> spam = {'colore': 'rosso', 'età': 42}
>>> for k, v in spam.items():
...     print('Chiave: ' + k + ' Valore: ' + str(v))
Chiave: età Valore: 42
Chiave: colore Valore: rosso
```

## Verificare se in un dizionario esistono una data chiave o un dato valore

Ricorderete dal [Capitolo 4](#) che gli **operatori** `in` e `not in` possono verificare se in una lista esiste un dato valore. Si possono usare questi operatori anche per vedere se in un dizionario **esistono una data chiave o un dato valore**. Inserite nella shell interattiva:

```
>>> spam = {'nome': 'Zophie', 'età': 7}
>>> 'nome' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'colore' in spam.keys()
False
>>> 'colore' not in spam.keys()
True
>>> 'colore' in spam
False
```

Nell’esempio, notate che ‘colore’ in `spam` è sostanzialmente una versione abbreviata di ‘colore’ in `spam.keys()`. Se si vuole verificare se un certo valore è (o meno) una chiave in un dizionario, si può semplicemente usare la parola chiave `in` (oppure `not in`) con il valore dizionario stesso.

## Il metodo `get()`

È noioso dover **verificare se in un dizionario esiste una chiave** prima di accedere al relativo valore. Per fortuna, i dizionari hanno un metodo `get()` che prende due argomenti: la chiave del valore da recuperare e un valore di “ripiego” (`fallback`) da restituire se la chiave non esiste.

Inserite nella shell interattiva:

```
>>> picnicItems = {'mele': 5, 'piatti': 2}
```

```
>>> 'Io porto ' + str(picnicItems.get('piatti', 0)) + ' piatti.'  
'Io porto 2 piatti.'  
>>> 'Io porto ' + str(picnicItems.get('uova', 0)) + ' uova.'  
'Io porto 0 uova.'
```

Poiché non esiste una chiave 'uova' nel dizionario `picnicItems`, il metodo `get()` restituisce il valore predefinito 0. Se non si fosse usato il metodo `get()`, il codice avrebbe provocato un messaggio di errore, come nell'esempio che segue:

```
>>> picnicItems = {'mele': 5, 'piatti': 2}  
>>> 'Io porto ' + str(picnicItems['uova']) + ' uova.'  
Traceback (most recent call last):  
  File "<pyshell#34>", line 1, in <module>  
    'Io porto ' + str(picnicItems['uova']) + ' uova.'  
KeyError: 'uova'
```

## Il metodo `setdefault()`

Spesso vi capiterà di dover **impostare un valore** in un dizionario **per una certa chiave** solo se quella chiave non ha già un valore. Il codice sarà analogo a questo:

```
spam = {'nome': 'Pooka', 'età': 5}  
if 'colore' not in spam:  
    spam['colore'] = 'nero'
```

Il metodo `setdefault()` offre un modo per farlo con una sola riga di codice. Il primo argomento passato al metodo è la chiave da controllare, il secondo argomento è il valore da attribuire a quella chiave se la chiave non esiste. Se la chiave esiste, il metodo `setdefault()` restituisce il valore della chiave. Inserite nella shell interattiva:

```
>>> spam = {'nome': 'Pooka', 'età': 5}  
>>> spam.setdefault('colore', 'nero')  
'black'  
>>> spam  
{'colore': 'nero', 'età': 5, 'nome': 'Pooka'}  
>>> spam.setdefault('colore', 'bianco')  
'nero'  
>>> spam  
{'colore': 'nero', 'età': 5, 'nome': 'Pooka'}
```

La prima volta che viene chiamato `setdefault()`, il dizionario in `spam` viene modificato in `{'colore': 'nero', 'età': 5, 'nome': 'Pooka'}`. Il metodo restituisce il valore 'nero' perché quello è ora il valore impostato per la chiave 'colore'. Quando poi viene chiamato `spam.setdefault('colore', 'bianco')`, il valore per quella chiave non viene cambiato in 'bianco', perché `spam` contiene già una chiave 'colore'.

Il metodo `setdefault()` è una buona scorciatoia per essere sicuri che esista una chiave. Ecco un breve programma che conta le occorrenze di ciascuna lettera in una stringa. Aprite il file editor e inserite il codice seguente, poi salvatelo come `characterCount.py`:

```
message = 'Era una fredda giornata di marzo e le campane suonavano mezzogiorno.'  
count = {}
```

```
for carattere in message:  
    count.setdefault(carattere, 0)  
    count[carattere] = count[carattere] + 1  
  
print(count)
```

Il programma cicla su ciascun carattere nella stringa della variabile `message`, contando quante volte compare ciascun carattere. Il metodo `setdefault()` garantisce che la chiave sia nel dizionario `count` (con valore di default 0), in modo che il programma non provochi un errore `KeyError` quando viene eseguito `count[carattere] = count[carattere] + 1`. Quando si esegue il programma, l'output sarà simile a questo:

```
{'.': 1, 'E': 1, 'u': 2, 'v': 1, 'c': 1, 'e': 5, 'T': 1, 'g': 2, 's': 1, 't': 1, 'n': 6, 'a': 10, 'd': 3, ' ': 10, 'r': 5, 'o': 7, 'l': 3, 'p': 1, 'z': 3, 'm': 3, 'f': 1}
```

Dall'output, si può vedere che la lettera `c` minuscola compare 1 volta, il carattere spazio compare 10 volte, la lettera `E` maiuscola compare 1 volta e così via. Il programma funzionerà qualunque sia la stringa nella variabile `message`, anche se dovesse essere lunga un milione di caratteri!

## Una stampa elegante (pretty printing)

Se importate nei vostri programmi il modulo `pprint`, avrete accesso alle funzioni  `pprint()` e `pformat()`, che provvedono al **pretty printing**, cioè alla stampa elegante dei valori di un dizionario, cosa utile per una presentazione degli elementi contenuti in un dizionario più pulita e comprensibile di quella fornita da `print()`. Modificate il precedente programma `characterCount.py` e salvatelo con il nome `prettyCharacterCount.py`.

```
import pprint  
message = 'Era una fredda giornata di marzo e le campane suonavano mezzogiorno.'  
count = {}
```

```
for carattere in message:  
    count.setdefault(carattere, 0)  
    count[carattere] = count[carattere] + 1
```

```
pprint pprint(count)
```

Questa volta, quando il programma viene eseguito l'output è molto più elegante e leggibile, con le chiavi ordinate:

```
{' ': 10,  
 '!': 1,  
 'E': 1,  
 'a': 10,  
 'c': 1,  
 'd': 3,  
 'e': 5,  
 'f': 1,  
 'g': 2,  
 'l': 3,  
 'T': 1,  
 'm': 3,  
 'n': 6,  
 'o': 7,
```

```
'p': 1,  
'r': 5,  
's': 1,  
't': 1,  
'u': 2,  
'v': 1,  
'z': 3}
```

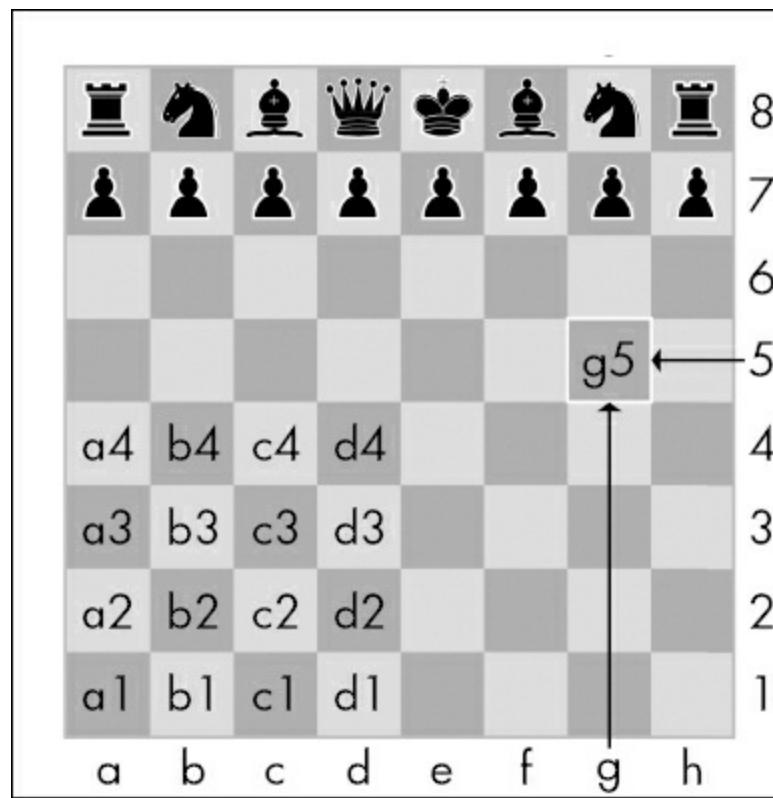
La funzione `pprint.pprint()` è particolarmente utile quando il dizionario contiene liste o dizionari annidati. Se volete ottenere il testo elegante come valore stringa invece di farlo visualizzare sullo schermo, chiamate `pprint.pformat()`. Queste due righe sono fra loro equivalenti:

```
pprint.pprint(qualcheValoreDiDizionario)  
print(pprint.pformat(qualcheValoreDiDizionario))
```

## Uso di strutture di dati come modelli di cose reali

Anche prima che esistesse Internet, era possibile **giocare a scacchi** con un avversario che si trovava dall'altra parte del mondo. Ciascun giocatore predisponeva a casa propria una scacchiera e poi i due a turno inviavano una cartolina all'altro descrivendo la propria mossa successiva. Per poterlo fare, i giocatori avevano bisogno di un modo per **descrivere** senza ambiguità lo **stato** della **scacchiera** e le loro **mosse**.

Nella notazione algebrica degli scacchi, le “case” sulla scacchiera sono identificate da un sistema di coordinate, un numero e una lettera, come nella [Figura 5.1](#).



**Figura 5.1** - Le coordinate di una scacchiera nella notazione algebrica degli scacchi.

I pezzi degli scacchi sono identificati da lettere maiuscole: *R* per il re, *D* per la regina, *T* per la torre, *A* per l’alfiere e *C* per il cavallo (nella notazione inglese le lettere sono, rispettivamente, *K*, *Q*, *R*, *B* e *N*). Una mossa si descrive utilizzando la lettera che identifica il pezzo e le coordinate della

destinazione. Una coppia di queste mosse descrive quello che succede a ogni turno (con il bianco che muove per primo); per esempio, la notazione 2. Cf3 Cc6 indica che al secondo turno il bianco ha spostato un cavallo in f3 e il nero un cavallo in c6.

La notazione algebrica presenta ancora qualche altra particolarità, ma il punto è che si può utilizzarla per descrivere senza ambiguità una partita di scacchi, senza dover essere per forza davanti a una scacchiera. Il vostro avversario può davvero trovarsi all'altro capo del mondo. Non è necessario nemmeno avere davanti i pezzi degli scacchi, se si ha una buona memoria: basta leggere le mosse comunicate via posta e aggiornare la scacchiera che si ha nella testa.

I computer hanno buona memoria. Un programma su un computer moderno può facilmente memorizzare miliardi di stringhe come '2. Cf3 Cc6'. È così che i computer possono giocare a scacchi senza avere una scacchiera fisica. Hanno un modello dei dati che rappresentano una scacchiera, e voi potete scrivere del codice che agisca su quel modello.

Qui possono entrare in scena liste e dizionari. Potete utilizzarli per costruire modelli di cose del mondo reale, come le scacchiere. Come primo esempio, usiamo un gioco un po' più semplice degli scacchi: il filetto (*tic-tac-toe*, in inglese).

## Una scacchiera di filetto

Una **scacchiera** per giocare a **filetto** assomiglia a un grande simbolo di diesis o cancelletto (#) con **nove caselle**, ciascuna delle quali può contenere una X, una O, o essere vuota. Per rappresentare la scacchiera con un **dizionario**, potete assegnare a ciascuna casella una stringa chiave-valore, come si vede nella [Figura 5.2](#).

'top-L'	'top-M'	'top-R'
'mid-L'	'mid-M'	'mid-R'
'low-L'	'low-M'	'low-R'

**Figura 5.2** - Le caselle di una scacchiera per il gioco del filetto con le chiavi corrispondenti.

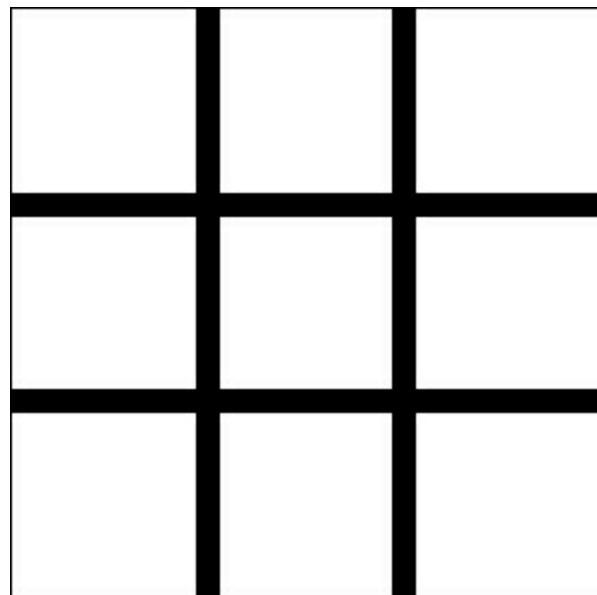
Potete usare valori stringa per rappresentare ciò che si trova in ciascuna casella della scacchiera: 'X', 'O' o '' (un carattere spazio). Dovete quindi memorizzare nove stringhe e per questo potete utilizzare un **dizionario** di valori. Il valore stringa con la chiave 'top-R' può rappresentare l'angolo superiore destro; il valore stringa con la chiave 'low-L' può rappresentare l'angolo inferiore sinistro; il valore stringa con la chiave 'mid-M' può rappresentare la casella centrale, e così via.

Questo dizionario è una struttura di dati che rappresenta una scacchiera di filetto. Memorizzate questa “scacchiera come dizionario” in una variabile `theBoard`. Aprite una nuova finestra di file editor e

Inserite il codice che segue, poi salvatelo con il nome *ticTacToe.py*:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',  
           'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',  
           'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

La struttura di dati memorizzata nella variabile `theBoard` rappresenta la scacchiera della [Figura 5.3](#).

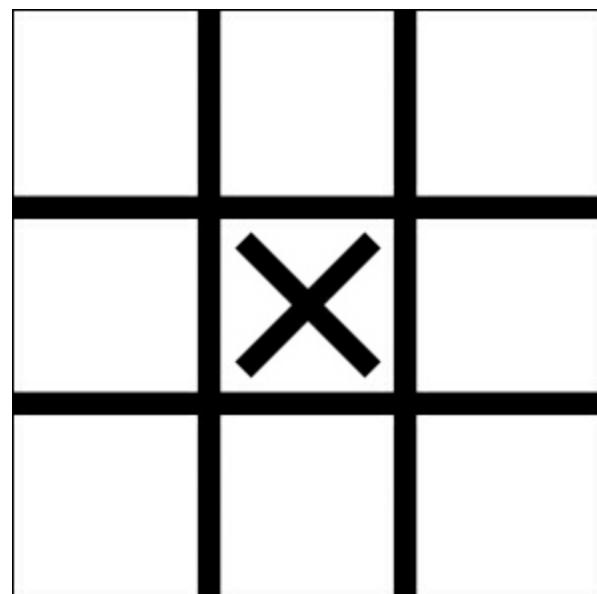


**Figura 5.3** - Una scacchiera di filetto vuota.

Dato che il valore di ciascuna chiave in `theBoard` è una stringa contenente un singolo spazio, questo dizionario rappresenta una scacchiera completamente vuota. Se il giocatore X muove per primo e sceglie la casella centrale, potreste rappresentare la scacchiera con questo dizionario:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',  
           'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',  
           'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

La struttura di dati in `theBoard` ora rappresenta la scacchiera della [Figura 5.4](#).

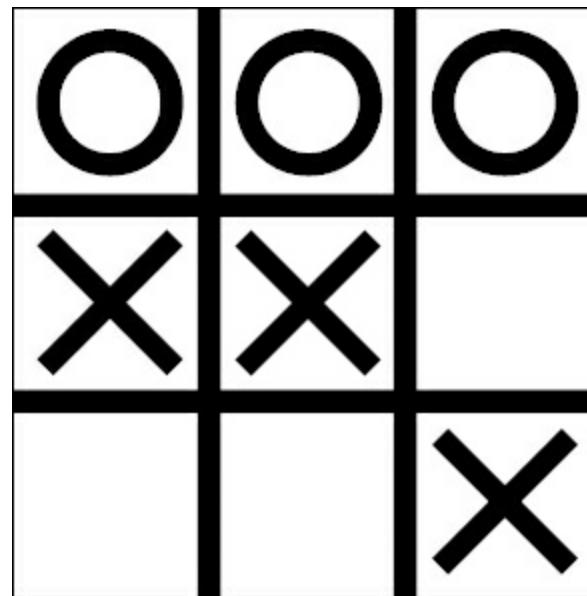


**Figura 5.4** - La prima mossa.

Una scacchiera su cui il giocatore O ha vinto, collocando tre O sulla riga superiore potrebbe essere rappresentata in questo modo:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

La struttura di dati in theBoard ora rappresenta la scacchiera della [Figura 5.5](#).



**Figura 5.5** - Il giocatore O vince.

Ovviamente, il giocatore vede quello che viene stampato sullo schermo, non i contenuti delle variabili. Proviamo a creare una funzione che stampi il dizionario della scacchiera. Aggiungete quanto segue a *ticTacToe.py* (il nuovo codice è in grassetto):

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + ' | ' + board['top-M'] + ' | ' + board['top-R'])
    print('---+---')
    print(board['mid-L'] + ' | ' + board['mid-M'] + ' | ' + board['mid-R'])
    print('---+---')
    print(board['low-L'] + ' | ' + board['low-M'] + ' | ' + board['low-R'])
printBoard(theBoard)
```

Quando eseguite questo programma, `printBoard()` stampa una scacchiera vuota.

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

La funzione `printBoard()` può trattare qualsiasi struttura di dati per il gioco del filetto che le si passi. Provate a modificare il codice così:

```

theBoard = {'top-L': '0', 'top-M': '0', 'top-R': '0', 'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}

def printBoard(board):
    print(board['top-L'] + ' | ' + board['top-M'] + ' | ' + board['top-R'])
    print(' -+--')
    print(board['mid-L'] + ' | ' + board['mid-M'] + ' | ' + board['mid-R'])
    print(' -+--')
    print(board['low-L'] + ' | ' + board['low-M'] + ' | ' + board['low-R'])
printBoard(theBoard)

```

Ora, quando eseguite questo programma, verrà stampata sullo schermo la nuova scacchiera.

```

0|0|0
-+-
X|X|
-+-
| |X

```

Poiché avete creato una struttura di dati che rappresenta una scacchiera per il filetto e avete scritto del codice in `printBoard()` per interpretare quella struttura di dati, ora avete un programma che è un “modello” della scacchiera.

Avreste potuto organizzare la struttura di dati in modo diverso (magari usando chiavi come 'TOP-LEFT' o 'alto-Sinistra' al posto di 'top-L'), ma, purché il codice funzioni con le vostre strutture di dati, avrete un programma che viene eseguito correttamente.

Per esempio, la funzione `printBoard()` si aspetta che la struttura di dati per il filetto sia un dizionario con nove chiavi per tutte le nove caselle. Se il dizionario che le passate non ha, poniamo, la chiave 'mid-L', il programma non funzionerà più.

```

0|0|0
-+-
Traceback (most recent call last):
  File "ticTacToe.py", line 10, in <module>
    printBoard(theBoard)
  File "ticTacToe.py", line 6, in printBoard
    print(board['mid-L'] + ' | ' + board['mid-M'] + ' | ' + board['mid-R'])
KeyError: 'mid-L'

```

Ora aggiungiamo del codice che permetta ai giocatori di indicare le proprie mosse. Modificate il programma `ticTacToe.py` in questo modo:

```

theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': ' ',
            'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + ' | ' + board['top-M'] + ' | ' + board['top-R'])
    print('---+---')
    print(board['mid-L'] + ' | ' + board['mid-M'] + ' | ' + board['mid-R'])
    print('---+---')
    print(board['low-L'] + ' | ' + board['low-M'] + ' | ' + board['low-R'])
turn = 'X'
for i in range(9):
    ❶   printBoard(theBoard)
    print('Tocca a ' + turn + '. In che casella vuoi andare?')
    ❷   move = input()
    ❸   theBoard[move] = turn
    ❹   if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
    printBoard(theBoard)

```

Questo nuovo codice stampa la scacchiera all'inizio di ciascun turno ❶, ottiene la mossa del giocatore attivo ❷, aggiorna di conseguenza la scacchiera di gioco ❸ e poi passa la mano all'altro giocatore ❹, prima di passare al turno successivo.

Quando si esegue questo programma, si otterrà qualcosa di simile a questo:

```
||  
-++-  
||  
-++-  
||  
Tocca a X. In che casella vuoi andare?  
mid-M
```

```
||  
-++-  
|X|  
-++-  
||  
Tocca a O. In che casella vuoi andare?  
low-L
```

```
||  
-++-  
|X|  
-++-  
0| |
```

--altre mosse--

```
0|0|X  
-++-  
X|X|0  
-++-  
0| |X
```

```
Tocca a X. In che casella vuoi andare?  
low-M  
0|0|X  
-++-  
X|X|0  
-++-  
0|X|X
```

Questo non è un gioco del filetto completo – per esempio, non controlla nemmeno se un giocatore abbia vinto – ma è sufficiente per vedere come si possano usare le strutture di dati nei programmi.

## NOTA

Se siete curiosi, il codice sorgente per un programma completo del gioco del filetto è descritto all'indirizzo <http://nostarch.com/automatestuff/>.

## Dizionari e liste annidati

Modellizzare una scacchiera per il gioco del filetto è stato piuttosto semplice: serviva solo un valore di dizionario con nove coppie chiave-valore. Quando si modellizzano cose più complicate, ci si trova anche ad aver bisogno di **dizionari** e **liste** che contengono altri dizionari e liste. Le liste sono utili per contenere una serie ordinata di valori, i dizionari per associare chiavi e valori. Per esempio, ecco un programma che usa un dizionario che contiene altri dizionari, per vedere chi porta che cosa per un picnic. La funzione `totalBrought()` può leggere questa struttura di dati e calcolare il numero totale

di oggetti di ciascun tipo portati da tutti i partecipanti.

```
allGuests = {'Alice': {'mele': 5, 'pretzel': 12},  
            'Bob': {'panini': 3, 'mele': 2},  
            'Carol': {'piatti': 3, 'torte di mele': 1}}  
  
def totalBrought(guests, item):  
    numBrought = 0  
❶    for k, v in guests.items():  
❷        numBrought = numBrought + v.get(item, 0)  
    return numBrought  
print('Numero di cose portate:')  
print(' - Mele ' + str(totalBrought(allGuests, 'mele')))  
print(' - Piatti ' + str(totalBrought(allGuests, 'piatti')))  
print(' - Brioches ' + str(totalBrought(allGuests, 'brioches')))  
print(' - Panini ' + str(totalBrought(allGuests, 'panini')))  
print(' - Torte di mele ' + str(totalBrought(allGuests, 'torte di mele')))
```

All'interno della funzione `totalBrought()`, il ciclo `for` itera sulle coppie chiave-valore in `guest` ❶. All'interno del ciclo, la stringa del nome del partecipante è assegnata a `k`, e il dizionario degli oggetti che porta al picnic è assegnato a `v`. Se il parametro `item` esiste come chiave in questo dizionario, a `numBrought` viene aggiunto il suo valore (la quantità) ❷. Se non esiste come chiave, il metodo `get()` restituisce 0 da aggiungere a `numBrought`.

L'output di questo programma è di questo tipo:

Numero di cose portate:

- Mele 7
- Piatti 3
- Brioches 0
- Panini 3
- Torte di mele 1

Può sembrare una cosa troppo semplice per darsi la pena di scrivere un programma che ne costituisca un modello, ma pensate che questa stessa funzione `totalBrought()` potrebbe tranquillamente gestire un dizionario che contenga migliaia di partecipanti, ciascuno dei quali porta migliaia di oggetti diversi. A quel punto avere tutte queste informazioni in una struttura di dati e la funzione `totalBrought()` vi farebbe risparmiare davvero una gran quantità di tempo.

Potete **modellizzare** cose con le **strutture di dati** nel modo che preferite, purché il resto del codice nel vostro programma possa funzionare correttamente con quel modello di dati. Essendo agli inizi della programmazione, non preoccupatevi troppo di trovare il modo “giusto” per modellizzare i dati. A mano a mano che acquisirete maggiore esperienza, escogiterete magari modelli più efficienti, ma la cosa importante è che il modello dei dati funzioni per le esigenze del vostro programma.

## Riepilogo

In questo capitolo avete scoperto tutto quello che c'è da sapere sui **dizionari**. Liste e dizionari sono valori che contengono molti valori, fra cui possono esserci anche altre liste e altri dizionari. I dizionari sono utili perché si può mettere in corrispondenza un elemento (la **chiave**) con un altro (il **valore**), mentre le liste contengono semplicemente una serie di valori in un ordine ben preciso. Si accede ai valori in un dizionario utilizzando le **parentesi quadre**, come si fa con le liste. Anziché un

indice intero, però, i dizionari possono avere chiavi di tipi di dati molto diversi: interi, in virgola mobile, stringhe o tuple. Organizzando i valori di un programma in **strutture di dati**, si possono creare rappresentazioni di oggetti del mondo reale: ne abbiamo visto un esempio con la scacchiera per giocare a filetto.

## Domande di ripasso

1. Come è fatto il codice per un dizionario vuoto?
2. Come sarebbe fatta una voce di dizionario che abbia come chiave 'foo' e come valore 42?
3. Qual è la differenza principale fra un dizionario e una lista?
4. Che cosa succede se si tenta di accedere a spam['foo'] se spam è {'bar': 100}?
5. Se un dizionario è memorizzato in spam, qual è la differenza fra le espressioni 'cat' in spam e 'cat' in spam.values()?
6. Qual è una scorciatoia per il codice seguente?

```
if 'colore' not in spam:  
    spam['colore'] = 'nero'
```

7. Quale modulo e quale funzione si possono usare per il “pretty printing” dei valori di un dizionario?

## Un po' di pratica

Per esercitarvi, scrivete dei programmi che svolgano queste attività.

## Oggetti posseduti in un gioco fantasy

State creando un videogioco fantasy. La struttura di dati per modellizzare l'insieme degli oggetti posseduti da un giocatore sarà un dizionario in cui le chiavi sono valori stringa che descrivono i singoli oggetti e il valore è un intero che precisa quanti oggetti di quel tipo possiede il giocatore. Per esempio, il valore dizionario {'fune': 1, 'torcia': 6, 'moneta oro': 42, 'pugnale': 1, 'freccia': 12} significa che il giocatore ha 1 fune, 6 torce, 42 monete d'oro e così via.

Scrivete una funzione `displayInventory()` che prenda ogni possibile oggetto e visualizzi l'elenco in un modo simile a questo:

Inventario:  
12 freccia  
42 moneta oro  
1 fune  
6 torcia  
1 pugnale

Numero totale di oggetti: 62

Suggerimento: potete usare un ciclo `for` e iterare su tutte le chiavi nel dizionario.

```
# inventory.py
stuff = {'fune': 1, 'torcia': 6, 'moneta oro': 42, 'pugnale': 1, 'freccia': 12}
def displayInventory(inventory):
    print("Inventario:")
    item_total = 0
    for k, v in inventory.items():
        # SCRIVETE VOI QUESTA PARTE
    print("Numero totale di oggetti: " + str(item_total))

displayInventory(stuff)
```

## Una funzione da lista a dizionario per gli oggetti nel gioco fantasy

Immaginate che il bottino ottenuto sconfiggendo un drago sia rappresentato da una lista di stringhe come questa:

```
dragoBottino = {'moneta oro', 'pugnale', 'moneta oro', 'moneta oro', 'rubino'}
```

Scrivete una funzione `addInventory(inventory, addedItems)`, in cui il parametro `inventory` sia un dizionario che rappresenta l'inventario degli oggetti in possesso del giocatore (come nel progetto precedente) e il parametro `addedItems` sia una lista come `dragoBottino`. La funzione `addInventory()` deve restituire un dizionario che rappresenta l'inventario aggiornato. Notate che la lista `addedItems` può contenere più esemplari dello stesso oggetto. Il codice dovrà essere qualcosa di simile a questo:

```
def addInventory(inventory, addedItems):
    # qui va il vostro codice

inv = {'moneta oro': 42, 'fune': 1}
dragoBottino = ['moneta oro', 'pugnale', 'moneta oro', 'moneta oro', 'rubino']
inv = addInventory(inv, dragoBottino)
displayInventory(inv)
```

Il programma precedente (con la vostra funzione `displayInventory()` ricavata dal progetto precedente) dovrà produrre questo output:

```
Inventario:
45 moneta oro
1 fune
1 rubino
1 pugnale
```

Numero totale di oggetti: 48

# Manipolare stringhe

Il **testo** è una delle forme più comuni dei dati che i vostri programmi gestiranno. Sapete già come **concatenare** due valori **stringa** con l'operatore `+`, ma sono molte altre le **operazioni possibili**.

Potete estrarre stringhe parziali da valori stringa, aggiungere o eliminare spazi, trasformare lettere da minuscole a maiuscole e verificare che le stringhe siano formattate correttamente. Potete addirittura scrivere codice Python che acceda agli Appunti per copiare e incollare testo. In questo capitolo imparerete queste cose e altre ancora. Poi affronterete due diversi progetti di programmazione: un semplice gestore di password e un programma per automatizzare un'attività noiosa come formattare brani di testo.

## Lavorare con le stringhe

Vediamo alcuni dei modi in cui Python consente di scrivere stringhe, stamparle e accedervi da codice.

### Letterali stringa

Scrivere valori stringa nel codice Python è decisamente semplice: iniziano e finiscono con un segno di apice. Ma come si può usare **un apice all'interno di una stringa**? Scrivere 'Questo è l'orologio di Marco' non funziona, perché Python pensa che la stringa finisca dopo la lettera `i` e che il resto ('orologio di Marco') sia codice non valido. Per fortuna, esistono molti modi per scrivere le stringhe.

#### Doppi apici

Le stringhe possono essere racchiuse anche fra **doppi apici** (anziché fra apici semplici). Un vantaggio dell'uso dei doppi apici è che la stringa può avere al suo interno un carattere apice singolo. Provate a inserire nella shell interattiva:

```
>>> spam = "Questo è l'orologio di Marco."
```

Dato che la stringa inizia con un doppio apice, Python sa che l'apice singolo fa parte della stringa e non ne indica invece la conclusione. Se però dovete usare all'interno di una stessa stringa sia apici singoli che doppi, dovete utilizzare i caratteri escape.

### **Caratteri escape**

Un **carattere escape** consente di utilizzare caratteri che altrimenti sarebbe impossibile inserire in una stringa. Un carattere escape è costituito da una barra retroversa (\) seguita dal carattere che si vuole aggiungere alla stringa, (Nonostante i caratteri siano due, di solito si parla di “un” carattere escape.) Per esempio, il carattere escape per l'apice singolo è \'. Potete utilizzarlo all'interno di una stringa racchiusa fra apici singoli. Per vedere come funzionano i caratteri escape, inserite nella shell interattiva:

```
>>> spam = 'Saluta l'amica di Giulia.'
```

Python sa che, poiché il simbolo di apice in \amica è preceduto dalla barra retroversa, non è un apice che indichi la conclusione della stringa. I caratteri escape \' e \" permettono di inserire apici singoli e doppi, rispettivamente, in qualsiasi stringa.

La [Tabella 6.1](#) elenca i caratteri escape che si possono utilizzare.

**Tabella 6.1** – I caratteri escape.

Carattere escape	Viene stampato come
\'	apice singolo
\"	doppio apice
\t	tabulazione
\n	newline (interruzione di riga, a capo)
\\\	barra retroversa

Inserite quanto segue nella shell interattiva:

```
>>> print("Salute a voi!\nChe tempo fa da voi?\nQui c'\\'è un bel sole.")
```

Salute a voi!  
Che tempo fa da voi?  
Qui c'è un bel sole.

### **Stringhe grezze (raw)**

Potete collocare una r prima del segno di apertura di una stringa, in modo da renderla una stringa grezza (**raw string**, in inglese). Una stringa grezza ignora completamente tutti i caratteri escape e stampa qualsiasi barra retroversa che compaia nella stringa. Per esempio, scrivete quanto segue nella shell interattiva:

```
>>> print(r'Questo è l'orologio di Marco.')
```

Questo è l'orologio di Marco.)

Poiché si tratta di una stringa grezza, Python tratta la barra retroversa come se fosse parte della stringa e non come simbolo iniziale di un carattere escape. Le stringhe grezze sono utili se dovete scrivere valori stringa che contengono molte barre retroverse, per esempio le stringhe utilizzate per le espressioni regolari, di cui parleremo nel prossimo capitolo.

## Stringhe su più righe con apici tripli

Si può usare il carattere escape `\n` per inserire un a capo (*newline*) in una stringa, ma spesso è più facile utilizzare **stringhe multiriga**. Una stringa multiriga in Python inizia e finisce con **tre apici singoli** o **tre apici doppi**. Apici, tabulazioni o a capo all'interno degli “apici tripli” sono considerati parte della stringa. Le regole dei rientri per i blocchi in Python non si applicano alle righe all'interno di una stringa multiriga.

Aprite il file editor e scrivete quanto segue:

```
print("Cara Alice,
```

il gatto di Carlo è stato arrestato per furto d'auto, vagabondaggio ed estorsione.

```
Con tutto l'affetto,  
Bob"
```

Salvate questo programma con il nome *catnapping.py* ed eseguitelo. L'output sarà fatto in questo modo:

Cara Alice,

il gatto di Carlo è stato arrestato per furto d'auto, vagabondaggio ed estorsione.

```
Con tutto l'affetto,  
Bob
```

Notate che il carattere apice singolo (d'auto, l'affetto) non deve essere un carattere escape. Nelle stringhe multiriga l'escape degli apici singoli e doppi è facoltativo. La chiamata di `print()` che segue stamperà lo stesso testo, ma non usa una stringa multiriga:

```
print('Cara Alice,\n\nIl gatto di Carlo è stato arrestato per furto d'auto, vagabondaggio ed estorsione.\n\nCon tutto l'affetto,\nBob')
```

## Commenti multiriga

Il carattere diesis (#) segna l'inizio di un commento per il resto della riga, ma spesso si usano **stringhe multiriga per commenti** che si estendono su più righe. Quello che segue è codice Python perfettamente valido:

```
"""Questo è un programma Python di prova,  
scritto da Al Sweigart al@inventwithpython.com
```

```
Questo programma è stato scritto per Python 3, non Python 2.  
"""
```

```
def spam()  
    """Questo è un commento multiriga per spiegare  
    che cosa fa la funzione spam()."""  
    print('Ciao!')
```

## Indicizzazione e slicing di stringhe

Sulle stringhe si possono usare **indici** e **sezioni** (*slice*) come sulle liste. Potete pensare la stringa 'Ciao mondo!' come una lista e ciascun carattere nella stringa come un elemento con un indice corrispondente.

```
' C   i   a   o   m   o   n   d   o   !   '  
 0   1   2   3   4   5   6   7   8   9   10
```

Lo spazio e il punto esclamativo sono inclusi nel conteggio dei caratteri, perciò 'Ciao mondo!' è lunga 11 caratteri, dalla C con indice 0 al ! con indice 10.

Inserite quanto segue nella shell interattiva:

```
>>> spam = 'Ciao mondo!'  
>>> spam[0]  
'C'  
>>> spam[e]  
'o'  
>>> spam[-1]  
'!'  
>>> spam[0:4]  
'Ciao'  
>>> spam[:4]  
'Ciao'  
>>> spam[5:]  
'mondo!'
```

Se specificate un indice, otterrete il carattere che si trova in quella posizione nella stringa. Se specificate un intervallo da un indice a un altro, l'indice iniziale è incluso, quello finale no. Per questo, se spam è 'Ciao mondo!', spam[0:4] è 'Ciao'. La sottostringa che si ottiene con spam[0:4] include tutto ciò che va da spam[0] a spam[3], escludendo lo spazio che si trova all'indice 4.

Notate che lo *slicing* di una stringa non modifica la stringa originale. Potete estrarre una sezione da una variabile e salvarla in un'altra variabile. Provate a scrivere quanto segue nella shell interattiva:

```
>>> spam = 'Ciao mondo!'  
>>> fizz = spam[0:4]  
>>> fizz  
'Ciao'
```

Effettuando una sezione e memorizzando la sottostringa risultante in un'altra variabile, si possono avere a disposizione sia la stringa intera sia la sottostringa, per poter accedere velocemente a entrambe.

## Gli operatori in e not in con le stringhe

Gli operatori `in` e `not in` possono essere utilizzati con le stringhe proprio come con i valori lista. Un'espressione con due stringhe connesse mediante `in` o `not in` avrà come valore un Booleano, True o False. Inserite quanto segue nella shell interattiva:

```
>>> 'Ciao' in 'Ciao mondo'  
True  
>>> 'Ciao' in 'Ciao'  
True  
>>> 'CIAO' in 'Ciao mondo'  
False  
>>> " in 'spam'  
True  
>>> 'gatti' not in 'cani e gatti'  
False
```

Queste espressioni controllano se la prima stringa (stringa esatta, considerando anche maiuscole e minuscole) è contenuta nella seconda.

## Metodi utili per le stringhe

Vari metodi permettono di analizzare stringhe o di creare valori stringa trasformati. Questa sezione descrive i metodi che vi capiterà più spesso di dover usare.

### I metodi `upper()`, `lower()`, `isupper()` e `islower()`

I metodi `upper()` e `lower()` per le stringhe restituiscono una nuova stringa in cui tutte le lettere della stringa originale sono state convertite in maiuscole o minuscole, rispettivamente. I caratteri contenuti nella stringa che non sono lettere rimangono immutati. Inserite quanto segue nella shell interattiva:

```
>>> spam = 'Ciao mondo!'  
>>> spam = spam.upper()  
>>> spam  
'CIAO MONDO!'  
>>> spam = spam.lower()  
>>> spam  
'ciao mondo!'
```

Notate che questi metodi non modificano la stringa stessa, ma restituiscono nuovi valori stringa. Se volete modificare la stringa originale, dovete chiamare i metodi `upper()` o `lower()` sulla stringa e assegnare poi la nuova stringa alla variabile in cui era memorizzata la stringa originale. Per questo dovete usare `spam = spam.upper()` per modificare la stringa in `spam` invece di usare semplicemente `spam.upper()`. (Del tutto analogamente, se la variabile `eggs` contiene il valore `10`, scrivere `eggs + 3` non muta il valore di `eggs`, ma `eggs = eggs + 3` sì.)

I metodi `upper()` e `lower()` sono utili se dovete effettuare un confronto senza tener conto della differenza fra minuscole e maiuscole. Le stringhe 'grande' e 'GRAnde' non sono uguali, ma nel piccolo programma che segue non importa se l'utente scrive Bene, BENE o beNE, perché la stringa prima del confronto viene convertita in tutte minuscole.

```
print('Come stai?')
feeling = input()
if feeling.lower() == 'bene':
    print('Anch'io sto bene.')
else:
    print('Spero che il resto della giornata ti vada meglio.')
```

Quando si esegue questo programma, viene visualizzata la domanda e l'inserimento di qualsiasi variante di `bene`, per esempio `BENe`, produce sempre l'output `Anch'io sto bene`. L'aggiunta ai vostri programmi di codice per gestire varianti o errori nell'input dell'utente, per esempio un uso non regolare delle maiuscole, li renderà più facili da usare e ridurrà le possibilità di cadute.

```
Come stai?
BENe
Anch'io sto bene.
```

I metodi `isupper()` e `islower()` restituiscono un valore Booleano `True` se la stringa ha almeno una lettera e tutte le lettere sono maiuscole o minuscole, rispettivamente. Altrimenti, il metodo restituisce `False`. Inserite quanto segue nella shell interattiva, e notate quello che restituisce ciascuna chiamata del metodo.

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

Poiché i metodi `upper()` e `lower()` restituiscono stringhe, si possono chiamare metodi stringa anche sui valori stringa restituiti.

Le espressioni risultanti appariranno come una catena di chiamate di metodi. Inserite quanto segue nella shell interattiva:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

## I metodi stringa `isX`

Oltre a `islower()` e `isupper()` esistono vari altri **metodi per le stringhe** il cui nome inizia con la parola `is`. Questi metodi restituiscono un valore **Booleano** che descrive la **natura della stringa**. Ecco alcuni fra i metodi `isX` più comuni.

- `isalpha()` restituisce True se la stringa è costituita solo da lettere e non è vuota.
- `isalnum()` restituisce True se la stringa consiste solo di lettere e numeri e non è vuota.
- `isdecimal()` restituisce True se la stringa è costituita solo da caratteri numeri e non è vuota.
- `isspace()` restituisce True se la stringa è costituita solo da spazi, tabulazioni e a capo e non è vuota.
- `istitle()` restituisce True se la stringa è costituita solo da parole che iniziano con una lettera maiuscola seguita da sole lettere minuscole.

Provate a inserire quanto segue nella shell interattiva.

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> ' '.isspace()
True
>>> 'Questo Può Essere Un Titolo'.istitle()
True
>>> 'Titolo 123'.istitle()
True
>>> 'Questo non è Un Titolo'.istitle()
False
>>> 'Ne anche QUESTO è un Titolo'.istitle()
False
```

I metodi stringa `isX` sono utili quando si deve **convalidare l'input** dell'utente. Per esempio, il programma seguente continua a chiedere agli utenti età e password, finché non forniscono un input valido. Aprite una nuova finestra di file editor e inserite questo programma, poi salvatelo con il nome `validateInput.py`:

```
while True:
    print('Inserisci la tua età:')
    age = input()
    if age.isdecimal():
        break
    print('Per indicare l'età devi scrivere un numero.')

while True:
    print('Scegli una nuova password (solo lettere e numeri):')
    password = input()
    if password.isalnum():
        break
    print('Le password devono essere formate da lettere e numeri solamente.')
```

Nel primo ciclo while, chiediamo all'utente la sua età e memorizziamo il suo input in age. Se age è un valore valido (decimale), usciamo da questo primo ciclo while e passiamo al secondo, che chiede una password; altrimenti, informiamo l'utente che deve inserire un numero e gli chiediamo di nuovo di inserire la sua età. Nel secondo ciclo while, chiediamo una password, memorizziamo l'input dell'utente in password e usciamo dal ciclo se l'input è alfanumerico; in caso contrario, non siamo soddisfatti e diciamo all'utente che la password deve essere alfanumerica, poi gli chiediamo nuovamente di inserire una password. Quando lo si esegue, l'output del programma può essere come quello seguente.

Inserisci la tua età:

**quarantadue**

Per indicare l'età devi scrivere un numero.

Inserisci la tua età:

**42**

Scegli una nuova password (solo lettere e numeri):

**secr3t!**

Le password devono essere formate da lettere e numeri solamente.

Scegli una nuova password (solo lettere e numeri):

**secr3t**

Chiamando isdecimal() e isalnum() su variabili, siamo in grado di controllare se i valori memorizzati in queste variabili sono decimali o meno, alfanumerici o meno. Qui queste verifiche ci aiutano a rifiutare l'input quarantadue e ad accettare 42, a rifiutare secr3t! e ad accettare secr3t.

## I metodi stringa startswith() e endswith()

I metodi startswith() e endswith() restituiscono True se il **valore stringa** su cui vengono chiamati **inizia** o (rispettivamente) **finisce** con la stringa passata al metodo; altrimenti restituiscono False. Inserite quanto segue nella shell interattiva:

```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

Questi metodi costituiscono utili alternative all'operatore di uguaglianza = se si deve verificare solo se la prima o l'ultima parte della stringa, anziché la stringa completa, è uguale a un'altra stringa.

## I metodi stringa join() e split()

Il metodo join() è utile quando si ha una lista di stringhe che devono essere **concatenate** in un unico valore stringa. Il metodo join() viene chiamato su una stringa, riceve una lista di stringhe e restituisce

una stringa. La stringa restituita è la concatenazione di tutte le stringhe nella lista passata. Per esempio, inserite nella shell interattiva:

```
>>> ', '.join(['gatti', 'topi', 'pipistrelli'])
'gatti, topi, pipistrelli'
>>> ' '.join(['Il', 'mio', 'nome', 'è', 'Simon'])
'Il mio nome è Simon'
>>> 'ABC'.join(['Il', 'mio', 'nome', 'è', 'Simon'])
'IlABCmioABCnomeABCèABCSimon'
```

Notate che la stringa su cui `join()` viene chiamata è inserita fra l'una e l'altra delle stringhe dell'argomento lista. Per esempio, quando si chiama `join(['gatti', 'topi', 'alci'])` sulla stringa `' '`, la stringa restituita è `'gatti, ratti, alci'`.

Ricordate che `join()` viene chiamato su un valore stringa e riceve un valore lista. (È facile sbagliarsi e fare il contrario.) Il metodo `split()` fa il contrario: viene chiamato su un valore stringa e restituisce una **lista di stringhe**. Inserite quanto segue nella shell interattiva:

```
>>> 'Il mio nome è Simon'.split()
['Il', 'mio', 'nome', 'è', 'Simon']
```

Per default, la stringa `'Il mio nome è Simon'` viene divisa nei punti in cui si trovano caratteri come uno spazio, una tabulazione o un segno di a capo. Questi caratteri “spazio” non sono inclusi nelle stringhe della lista restituita. Potete passare al metodo `split()` una stringa di delimitazione per specificare che deve suddividere la stringa su cui viene chiamata nei punti in cui occorre la stringa passata. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> 'IlABCmioABCnomeABCèABCSimon'.split('ABC')
['Il', 'mio', 'nome', 'è', 'Simon']
>>> 'Il mio nome è Simon'.split('m')
['Il ', 'io no', 'e è Si', 'on']
```

Si usa spesso `split()` per suddividere una **stringa multiriga**, separandola dove compaiono i caratteri di a capo. Inserite quanto segue nella shell interattiva:

```
>>> spam = """Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".
```

**Please do not drink it.**

**Sincerely,**

**Bob""**

```
>>> spam.split('\n')
['Dear Alice', 'How have you been? I am fine.', 'There is a container in the fridge', 'that is labeled "Milk Experiment".', "", 'Please do not drink it.', 'Sincerely', 'Bob']
```

Passando a `split()` come argomento `\n`, la stringa multiriga memorizzata in `spam` viene divisa in corrispondenza degli a capo e viene restituita una lista in cui ciascun elemento corrisponde a una riga della stringa.

## Giustificare il testo con `rjust()`, `ljust()` e `center()`

I metodi `stringa.rjust()` e `ljust()` restituiscono una versione della stringa su cui vengono chiamati che è “imbottita” di spazi in modo da **giustificare il testo**. Il primo argomento di entrambi i metodi è un intero che è la lunghezza della stringa giustificata. Inserite quanto segue nella shell interattiva:

```
>>> 'Hello'.rjust(10)
'      Hello'
>>> 'Hello'.rjust(20)
'          Hello'
>>> 'Hello World'.rjust(20)
'          Hello World'
>>> 'Hello'.ljust(10)
'Hello      '
```

`'Hello'.rjust(0)` dice che vogliamo allineare a destra 'Hello' in una stringa di lunghezza totale 10. 'Hello' è formata da cinque caratteri, perciò alla sua sinistra verranno aggiunti cinque spazi, il che ci darà una stringa di 10 caratteri, con 'Hello' allineato a destra.

Un secondo argomento facoltativo per `rjust()` e `ljust()` specifica un **carattere di riempimento** diverso dal carattere spazio. Inserite quanto segue nella shell interattiva:

```
>>> 'Hello'.rjust(20, '*')
*****Hello*****
>>> 'Hello'.ljust(20, '-')
'Hello-----'
```

Il metodo `center()` funziona come i precedenti, ma centra il testo anziché allinearla a sinistra o a destra. Inserite quanto segue nella shell interattiva:

```
>>> 'Hello'.center(20)
'      Hello
>>> 'Hello'.center(20, '=')
'=====Hello====='
```

Questi metodi sono particolarmente utili quando si devono **stampare dati in forma tabulare** con le spaziature giuste. Aprite una nuova finestra di file editor e inserite il codice seguente, poi salvatelo con il nome `picnicTable.py`:

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PER IL PICNIC'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))
picnicItems = {'panini': 4, 'mele': 12, 'piatti': 4, 'biscotti': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

In questo programma, definiamo un metodo `printPicnic()` che accetta un dizionario di informazioni e usa `center()`, `ljust()` e `rjust()` per visualizzare quelle informazioni in un formato di tabella ben allineata.

Il dizionario che passiamo a `printPicnic()` è `picnicItems`. In `picnicItems`, abbiamo 4 panini, 12 mele, 4 piatti e 8000 biscotti. Vogliamo organizzare queste informazioni in due colonne, con il nome di ciascun tipo di oggetto sulla sinistra e la quantità sulla destra. A questo scopo, decidiamo quanto vogliamo che siano larghe le colonne di sinistra e di destra. Insieme con il nostro dizionario, passeremo a `printPicnic()` anche questi due valori.

`printPicnic()` prende in ingresso un dizionario, `leftWidth` per la larghezza della colonna di sinistra di una tabella e `rightWidth` per la larghezza della colonna di destra. Stampa un titolo, PER IL PICNIC, centrato sopra la tabella. Poi cicla sul dizionario, stampa ogni coppia chiave-valore su una riga a sé con la chiave allineata a sinistra, completata con caratteri di riempimento punto e il valore allineato a destra e completato da spazi.

Una volta definita `printPicnic()`, definiamo il dizionario `picnicItems` e chiamiamo due volte `printPicnic()`, passandole larghezze diverse per le due colonne.

Quando si esegue il programma, gli oggetti per il picnic vengono presentati due volte. La prima volta la colonna di sinistra è larga 12 caratteri, quella di destra 5; la seconda volta le loro larghezze sono rispettivamente di 20 e 6 caratteri.

```
--PER IL PICNIC--  
piatti..... 4  
biscotti.... 8000  
mele..... 12  
panini..... 4  
-----PER IL PICNIC-----  
piatti..... 4  
biscotti.... 8000  
mele..... 12  
panini..... 4
```

Utilizzando `rjust()`, `ljust()` e `center()` si può fare in modo che le stringhe vengano allineate in modo elegante, anche se non si sa esattamente da quanti caratteri sono costituite le singole stringhe.

## Eliminare spazi bianchi con `strip()`, `rstrip()` e `lstrip()`

A volte si vogliono **eliminare i caratteri spazio** bianco (spazi, tabulazioni e a capo) a sinistra, a destra o da entrambi gli estremi di una stringa. Il metodo `strip()` restituisce una nuova stringa senza alcun carattere spazio all'inizio o alla fine. I metodi `lstrip()` e `rstrip()` eliminano i caratteri spazio a sinistra e a destra, rispettivamente.

Inserite quanto segue nella shell interattiva:

```
>>> spam = 'Hello World'  
>>> spam.strip()  
'Hello World'  
>>> spam.lstrip()  
'Hello World'  
>>> spam.rstrip()  
'Hello World'
```

Facoltativamente, un argomento stringa specificherà quali caratteri debbano essere eliminati dalle estremità della stringa. Inserite quanto segue nella shell interattiva:

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'  
>>> spam.strip('ampS')  
'BaconSpamEggs'
```

Passando a `strip()` l'argomento 'ampS' gli si dice di eliminare ogni occorrenza di a, m, p e una s maiuscola dalle estremità della stringa memorizzata in `spam`. L'ordine dei caratteri nella stringa passata a `strip()`

non è importante: `strip('ampS')` si comporterà in modo identico a `strip('mapS')` e a `strip('Spam')`.

## Copiare e incollare stringhe con il modulo pyperclip

Il modulo `pyperclip` possiede funzioni `copy()` e `paste()` che possono inviare testo agli Appunti del vostro computer e, rispettivamente, ricevere testo dagli Appunti. L'invio dell'output del programma agli Appunti renderà facile incollarlo in un messaggio di posta elettronica, in un word processor o in qualche altro tipo di software.

Pyperclip non è fornito insieme a Python. Per installarlo, seguite le indicazioni per l'installazione di moduli di terze parti riportate nell'[Appendice A](#). Dopo aver installato il modulo `pyperclip`, inserite quanto segue nella shell interattiva:

```
>>> import pyperclip  
>>> pyperclip.copy('Ciao mondo!')  
>>> pyperclip.paste()  
'Ciao mondo!'
```

Ovviamente, se qualcosa all'esterno del vostro programma dovesse modificare i contenuti degli Appunti, la funzione `paste()` restituirebbe i nuovi contenuti. Per esempio, se copiassi questa frase negli Appunti e poi chiamassi `paste()`, il risultato sarebbe questo:

```
>>> pyperclip.paste()  
' Per esempio, se copiassi questa frase negli Appunti e poi chiamassi paste(), il risultato sarebbe questo.'
```

### Eseguire script Python fuori da IDLE

Fin qui, avete eseguito i vostri script di Python utilizzando la shell interattiva e l'editor di file in IDLE, ma non sempre vorrete passare attraverso l'apertura di IDLE e dello script Python ogni volta che volete eseguire uno script.

Per fortuna, esistono scorciatoie che potete seguire per semplificare l'esecuzione di script Python. I passi sono leggermente diversi per Windows, OS X e Linux, ma sono tutti descritti nell'[Appendice B](#). Leggete l'[Appendice B](#) per scoprire come eseguire comodamente i vostri script Python e come passare argomenti da riga di comando. (Non è possibile invece passare argomenti da riga di comando ai vostri programmi, se utilizzate IDLE.)

## Progetto: password locker

Probabilmente avrete account per molti siti web diversi. È una cattiva abitudine usare la stessa password per tutti perché, se uno di questi siti subisse una violazione, gli hacker conoscerebbero la password per tutti gli altri vostri account. È meglio usare un software gestore di password che usi una password principale (o “master”) per sbloccare il sistema di gestione delle password. A quel punto potete copiare la password di ciascun account negli Appunti e incollarla nel campo *Password* del sito.

Il programma gestore di password che creerete in questo esempio non è sicuro, ma offre una dimostrazione essenziale di come funzionino i programmi di questo tipo.

### I progetti di capitolo

Questo è il primo “progetto di capitolo” del libro. Da qui in poi, ogni capitolo presenterà dei progetti che dimostrano i concetti trattati

in quel capitolo. I progetti sono scritti in modo da portarvi da una finestra vuota del file editor fino a un programma completo e funzionante.  
Come per gli esempi della shell interattiva, non limitatevi a leggere i paragrafi che riguardano i progetti: seguiteli passo per passo sul vostro computer!

## Passo 1: progetto del programma e strutture di dati

Vorrete poter eseguire questo programma con un argomento da riga di comando che sia il nome dell'account, per esempio *email* o *blog*. Potrete poi copiare la password di quell'account negli Appunti in modo che l'utente poi la possa incollare in un campo *Password*. In questo modo l'utente può avere password lunghe e complicate senza doverle memorizzare.

Aprite una nuova finestra di file editor e salvate il programma con il nome *pw.py*. Dovete iniziare il programma con una riga `#!` (in gergo si chiama “riga shebang”, vedete l'[Appendice B](#)) e dovete scrivere un commento che descriva brevemente il programma. Poiché volete associare il nome di ciascun account alla relativa password, potete memorizzare le password come stringhe in un dizionario. Il dizionario sarà la struttura di dati che organizza i dati relativi ai vostri account e alle vostre password. Iniziate il programma in un modo simile a questo:

```
#!/usr/bin/python3
# pw.py - Un programma password locker non sicuro.

PASSWORDS = {'email': 'F7min1BDDuvMJuxESSKHFhTxFtjVB6',
              'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdtn',
              'valigia': '12345'}
```

## Passo 2: gestire gli argomenti da riga di comando

Gli argomenti da riga di comando saranno memorizzati nella variabile `sys.argv`. (Vedete l'[Appendice B](#) per maggiori informazioni su come usare gli argomenti da riga di comando nei vostri programmi.) Il primo elemento nella lista `sys.argv` dovrà sempre essere una stringa che contiene il nome di file del programma ('*pw.py*') e il secondo elemento dovrà essere il primo argomento da riga di comando. Per questo programma, questo argomento è il nome dell'account di cui si vuole la password. Dato che l'argomento da riga di comando è obbligatorio, presenterete all'utente un messaggio per ricordarglielo, nel caso se ne dimenticasse (cioè se la lista `sys.argv` ha meno di due valori). Ecco come modificare il programma:

```
#!/usr/bin/python3
# pw.py - Un programma password locker non sicuro.

PASSWORDS = {'email': 'F7min1BDDuvMJuxESSKHFhTxFtjVB6',
              'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdtn',
              'valigia': '12345'}

import sys
if len(sys.argv) < 2:
    print("Uso: python pw.py [account] - copia la password dell'account")
    sys.exit()

account = sys.argv[1] # il primo argomento da riga di comando è il nome dell'account
```

## Passo 3: copiare la password giusta

Ora che il nome dell'account è memorizzato come stringa nella variabile `account`, dovete vedere se esiste come chiave nel dizionario `PASSWORDS`. Se sì, ne copierete il valore negli Appunti utilizzando `pyperclip.copy()`. (Poiché usate il modulo `pyperclip`, dovete importarlo.) Notate che non è strettamente *necessario* usare la variabile `account`; potreste anche usare `sys.argv[1]` ovunque in questo programma si usa `account`. Ma una variabile che si chiama `account` è molto più leggibile, rispetto a un'espressione un po' misteriosa come `sys.argv[1]`.

Modificate il vostro programma in questo modo:

```
#!/usr/bin/python3
# pw.py - Un programma password locker non sicuro.

PASSWORDS = {'email': 'F7min1BDDuvMJuxESSKHFhTxFtjVB6',
              'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sd',
              'valigia': '12345'}

import sys, pyperclip
if len(sys.argv) < 2:
    print("Uso: python pw.py [account] - copia la password dell'account")
    sys.exit()

account = sys.argv[1] # il primo argomento da riga di comando è il nome dell'account

if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])
    print('Password per ' + account + ' copiata negli Appunti.')
else:
    print('Non esiste alcun account con il nome ' + account)
```

Questo nuovo pezzo di codice cerca nel dizionario `PASSWORDS` il nome dell'account. Se il nome dell'account è una chiave nel dizionario, prende il valore corrispondente, lo copia negli Appunti e stampa un messaggio in cui dice di aver copiato il valore. In caso contrario, stampa un messaggio con cui precisa che non esiste account con quel nome.

Questo è lo script completo. Utilizzando le istruzioni contenute nell'[Appendice B](#) per lanciare facilmente programmi da riga di comando, ora avete un modo rapido per copiare le password dei vostri account negli Appunti. Dovrete modificare nel codice sorgente il valore del dizionario `PASSWORDS` ogni volta che volete aggiornare il programma con una nuova password.

Ovviamente, è probabile che non vogliate conservare tutte le vostre password in un singolo luogo, da dove chiunque potrebbe facilmente copiarle. Ma potete modificare questo programma e usarlo per copiare rapidamente frammenti di testo negli Appunti. Supponiamo che dobbiate spedire molti messaggi di posta elettronica che abbiano in comune molti paragrafi standard. Potete mettere ciascun paragrafo come valore nel dizionario `PASSWORDS` (probabilmente a questo punto vorrete anche cambiare il nome del dizionario) e poi avrete un modo rapido per selezionare e copiare uno dei molti paragrafi standard negli Appunti.

In Windows, potete creare un file batch per eseguire questo programma con la finestra *Eseguì* chiamata con Windows-R. (Per maggiori informazioni sui file batch, consultate l'[Appendice B](#).) Scrivete quanto segue nel file editor e salvate il file con il nome `pw.bat` nella cartella `C:\Windows`:

```
@py.exe C:\Python34\pw.py %*
```

Creato questo file batch, per eseguire il programma per le password in Windows basterà premere Windows-R e scrivere pw <nome dell'account>.

## Progetto: aggiungere punti elenco al markup wiki

Quando si modifica un articolo di Wikipedia, si può creare un elenco puntato scrivendo ciascuna voce dell'elenco su una propria riga e poi scrivendo a inizio riga un asterisco. Poniamo però che abbiate un elenco davvero molto lungo e che vogliate aggiungere l'asterisco iniziale a ciascuna voce: potreste semplicemente scrivere un asterisco all'inizio di ogni singola riga, procedendo una riga dopo l'altra, ma potreste anche automatizzare questo compito con un breve script Python.

Lo script *bulletPointAdder.py* prenderà il testo dagli Appunti, aggiungerà all'inizio di ogni riga un asterisco e uno spazio e poi incollerà il nuovo testo negli Appunti. Per esempio, se copiassi negli Appunti questo testo (per l'articolo di Wikipedia “List of Lists of Lists”):

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

e poi eseguissi il programma *bulletPointAdder.py*, poi gli Appunti conterrebbero:

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

## Passo 1: copiare e incollare dagli Appunti

Volete che il programma *bulletPointAdder.py* faccia queste cose:

1. incolla testo dagli Appunti;
2. faccia qualcosa con quel testo;
3. copi il nuovo testo negli Appunti.

Il secondo passo è un po' più complicato, ma i passi 1 e 3 sono immediati: coinvolgono semplicemente le funzioni `pyperclip.copy()` e `pyperclip.paste()`. Per il momento, scriviamo la parte del programma che riguarda i passi 1 e 3.

Inserite quanto segue e salvate il programma con il nome *bulletPointAdder.py*:

```
#!/usr/bin/python3
# bulletPointAdder.py - Aggiunge punti elenco per Wikipedia
# all'inizio di ogni riga di testo negli Appunti.
```

```
import pyperclip
text = pyperclip.paste()

# DA FARE: Separare le righe e aggiungere gli asterischi.

pyperclip.copy(text)
```

Il commento DA FARE è un promemoria: alla fine bisognerà completare anche questa parte del programma. Il passo successivo è proprio questo.

## Passo 2: separare le righe di testo e aggiungere l'asterisco

La chiamata di `pyperclip.paste()` restituisce tutto il testo presente negli Appunti come una lunga stringa. Se usiamo l'esempio della “List of Lists of Lists”, la stringa memorizzata sarrebbe come questa:

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author abbreviation\nLists of cultivars'
```

I caratteri `\n` di a capo in questa stringa fanno sì che venga visualizzata su più righe quando viene stampata o incollata dagli Appunti. Vi sono molte “righe” in quest'unico valore stringa. Volete aggiungere un asterisco all'inizio di ciascuna riga.

Potreste scrivere del codice che cerchi ciascun carattere `\n` di a capo nella stringa e poi aggiunga l'asterisco subito dopo, ma sarebbe più facile usare il metodo `split()` per avere una lista di stringhe, una per ciascuna riga nella stringa originale, poi aggiungere l'asterisco davanti a ciascuna stringa nella lista.

Modificate il vostro programma in questo modo:

```
#!/usr/bin/python3
# bulletPointAdder.py - Aggiunge punti elenco per Wikipedia
# all'inizio di ogni riga di testo negli Appunti.

import pyperclip
text = pyperclip.paste()

# Separare le righe e aggiunge gli asterischi.
lines = text.split('\n')
for i in range(len(lines)):
    lines[i] = '* ' + lines[i] # aggiunge l'asterisco a ciascuna stringa in "lines"

pyperclip.copy(text)
```

Suddividiamo il testo in corrispondenza degli a capo per avere una lista in cui ciascun elemento è una riga del testo. Memorizziamo la lista in `lines` e poi cicliamo sugli elementi di `lines`. Per ciascuna riga, aggiungiamo all'inizio della riga un asterisco e uno spazio. Ora ogni stringa di `lines` inizia con un asterisco.

## Passo 3: concatenare le righe modificate

La lista `lines` ora contiene le righe modificate che iniziano con un asterisco e uno spazio, ma `pyperclip.copy()` si aspetta un singolo valore stringa, non una lista di valori stringa. Per creare il valore stringa singolo, passiamo `lines` al metodo `join()` per avere una stringa che sia la concatenazione delle stringhe della lista. Modificate il vostro programma in questo modo:

```
#!/usr/bin/python3
# bulletPointAdder.py - Aggiunge punti elenco per Wikipedia
# all'inizio di ogni riga di testo negli Appunti.

import pyperclip
```

```
text = pyperclip.paste()

# Separa le righe e aggiunge gli asterischi.
lines = text.split('\n')
for i in range(len(lines)): # cicla su tutti gli indici della lista lines"
    lines[i] = '*' + lines[i] # aggiunge l'asterisco a ciascuna stringa in "lines"
text = '\n'.join(lines)
pyperclip.copy(text)
```

Quando il programma viene eseguito, sostituisce il testo presente negli Appunti con un testo che ha un asterisco all'inizio di ogni riga. Ora il programma è completo, e potete metterlo alla prova eseguendolo con testo copiato negli Appunti.

Anche se non avete bisogno di automatizzare questo compito specifico, potreste voler automatizzare qualche altro tipo di manipolazione del testo, per esempio eliminare gli spazi in eccesso a fine riga o convertire il testo in maiuscole o minuscole. Qualsiasi cosa vi serva, potete usare gli Appunti per l'input e l'output.

## Riepilogo

Il **testo** è una forma comune di dati e Python mette a disposizione molti **metodi stringa** utili per elaborare il testo memorizzato in valori stringa. Farete uso dell'indicizzazione delle sezioni e dei metodi stringa in quasi tutti i programmi che scriverete in Python.

I programmi che state scrivendo per ora non sembreranno molto raffinati: non hanno interfacce grafiche con immagini e testo colorato. Fin qui, avete visualizzato il testo con `print()` e chiesto all'utente di inserire testo con `input()`. L'utente però può inserire rapidamente grandi quantità di testo attraverso gli Appunti. Questa possibilità apre una strada utile per la scrittura di programmi che manipolino quantità elevate di testo. Questi programmi basati sul testo non avranno forse finestre o grafica allettante, ma possono svolgere molto lavoro utile in breve tempo.

Un altro modo per manipolare grandi quantità di testo consiste nel leggere e scrivere file direttamente dal disco fisso. Imparerete come si fa nel prossimo capitolo.

Quel che abbiamo visto fin qui esaurisce i concetti fondamentali della programmazione in Python! Continuerete ad apprendere nuovi concetti nel resto del libro, ma ora ne sapete abbastanza da cominciare a scrivere programmi utili per automatizzare delle attività. Forse pensate di non sapere abbastanza di Python da fare cose come scaricare pagine web, aggiornare fogli di calcolo o spedire messaggi di testo, ma per questo vengono in soccorso i moduli Python. Questi moduli, scritti da altri programmatore, mettono a disposizione funzioni che rendono facile fare tutte queste cose. Vediamo dunque come scrivere programmi reali che svolgano attività automatizzate utili.

## Domande di ripasso

1. Che cosa sono i caratteri escape?
2. Che cosa rappresentano i caratteri escape `\n` e `\t`?
3. Come si può inserire un carattere di barra retroversa `\` in una stringa?
4. Il valore stringa "Howl's Moving Castle" è una stringa valida. Perché quel carattere apice in `Howl's` non è un problema, anche se non è un carattere escape?
5. Se non volete inserire il carattere `\n` nella vostra stringa, come potete scrivere una stringa che contenga degli a capo?

**6.** Qual è il valore delle espressioni seguenti?

- 'Ciao mondo![1]
- 'Ciao mondo![0:4]
- 'Ciao mondo![4]
- 'Ciao mondo![3:]

**7.** Qual è il valore delle espressioni seguenti?

- 'Ciao'.upper()
- 'Ciao'.upper().isupper()
- 'Ciao'.upper().lower()

**8.** Qual è il valore delle espressioni seguenti?

- 'Ricorda, ricorda, il cinque di Novembre.'.split()
- '-'.join('Ce ne può essere solo uno.'.split())

**9.** Quali metodi stringa potete usare per allineare a destra, a sinistra o al centro una stringa?

**10.** Come potete eliminare i caratteri spazio dall'inizio o dalla fine di una stringa?

## Un po' di pratica

Per esercitarvi, scrivete un programma che svolga l'attività seguente.

## Stampa tabella

Scrivete una funzione `printTable()` che prenda una lista di liste di stringhe e la visualizzi sotto forma di tabella ben organizzata con ciascuna colonna allineata a destra. Ipotizzate che tutte le liste interne contengano lo stesso numero di stringhe. Per esempio, il valore potrebbe essere come questo:

```
tableData = [['apples', 'oranges', 'cherries', 'banana'],
             ['Alice', 'Bob', 'Carol', 'David'],
             ['dogs', 'cats', 'moose', 'goose']]
```

La vostra funzione `printTable()` stamperà questo:

```
    apples Alice  dogs
    oranges   Bob   cats
cherries   Carol moose
      banana David goose
```

Suggerimento: il vostro codice dovrà prima trovare la stringa più lunga in ciascuna delle liste interne, in modo che la colonna corrispondente sia larga abbastanza da contenere tutte le stringhe. Potete memorizzare la larghezza massima di ciascuna colonna come una lista di interi. La funzione `printTable()` può iniziare con `colWidths = [0] * len(tableData)`, che creerà una lista contenente tanti valori 0 quante sono le liste interne in `tableData`. In questo modo, `colWidths[0]` può memorizzare la larghezza della stringa più lunga in `tableData[0]`, `colWidths[1]` può conservare la larghezza della stringa più lunga in `tableData[1]` e così via. Potete poi trovare il valore più grande nella lista `colWidths` per stabilire quale larghezza intera passare al metodo `stringa.rjust()`.

## **Parte 2**

### **Automatizzare attività**

# Trovare corrispondenze con le espressioni regolari

Vi sarà sicuramente capitato di **cercare** del testo premendo Ctrl-F e poi scrivendo le parole da trovare. Le **espressioni regolari** permettono di fare un passo in più: consentono di specificare uno **schema** di testo da cercare.

Magari non sapete il numero telefonico preciso di un'azienda ma, se vivete negli USA o in Canada, sapete che è formato da tre cifre, un trattino e quattro altre cifre (e, facoltativamente, da un prefisso di tre cifre). Così noi umani lo riconosciamo quando lo vediamo: 415-555-1234 è un numero di telefono, 4,155,551,234 non lo è.

Le espressioni regolari sono utili ma pochi, se non programmano, ne conoscono l'esistenza, benché quasi tutti gli editor di testo e i word processor, da Microsoft Word a OpenOffice Writer, abbiano funzioni di ricerca e sostituzione che accettano le espressioni regolari. Queste possono far risparmiare molto tempo, non solo a chi usa il software, ma anche ai programmatore. Lo scrittore Cory Doctorow sostiene addirittura che, prima ancora che a programmare, bisogna insegnare a usare le espressioni regolari:

Conoscere [le espressioni regolari] può significare la differenza fra risolvere un problema in 3 passi e risolverlo in 3000 passi. Se sei un nerd, ti dimentichi che i problemi che tu risolvi schiacciando un paio di tasti alle altre persone possono richiedere giornate di lavoro noioso e in cui è facile sbagliare.<sup>1</sup>

In questo capitolo, comincerete a scrivere un programma per trovare schemi di testo *senza* usare le espressioni regolari, poi vedrete come l'uso delle espressioni regolari renda il codice molto meno farraginoso. Vi mostrerò la ricerca semplice di corrispondenze mediante espressioni regolari, poi passeremo a qualche caratteristica più avanzata, come la sostituzione di stringhe e la creazione di classi di caratteri personalizzate. Infine, al termine del capitolo, scriverete un programma che può estrarre automaticamente da un blocco di testo numeri telefonici e indirizzi email.

## Trovare schemi di testo senza le espressioni regolari

Supponiamo che vogliate trovare un numero telefonico (americano o canadese) in una stringa. Conoscete lo schema: tre numeri, un trattino, tre numeri, un trattino, quattro numeri; per esempio, 415-555-4242.

Creiamo una funzione `isPhoneNumber()` per verificare se una stringa corrisponda a questo schema: il valore restituito sarà `True` o `False`.

Aprite una nuova finestra di file editor e scrivete il codice seguente, poi salvate il file con il nome `isPhoneNumber.py`:

```
❶ def isPhoneNumber(text):
❷     if len(text) != 12:
❸         return False
❹     for i in range(0, 3):
❺         if not text[i].isdecimal():
❻             return False
❼     if text[3] != '-':
⽿         return False
⽾     for i in range(4, 7):
⽿         if not text[i].isdecimal():
⽽             return False
⽾     if text[7] != '-':
⽽         return False
⽾     for i in range(8, 12):
⽿         if not text[i].isdecimal():
⽽             return False
⽾     return True

print('415-555-4242 è un numero telefonico:')
print(isPhoneNumber('415-555-4242'))
print('Moshi moshi è un numero telefonico:')
print(isPhoneNumber('Moshi moshi'))
```

Quando questo programma viene eseguito, l'output è simile a questo:

415-555-4242 è un numero telefonico:

True

Moshi moshi è un numero telefonico:

False

La funzione `isPhoneNumber()` contiene codice che effettua vari controlli per stabilire se una stringa nella variabile `text` è un numero telefonico valido. Se tutti i controlli falliscono, la funzione restituisce `False`. Prima il codice verifica che la stringa sia esattamente lunga 12 caratteri ❶. Poi controlla che il prefisso (cioè i primi tre caratteri in `text`) sia formato solo da caratteri numerici ❷. Il resto della funzione controlla che la stringa segua lo schema di un numero telefonico. Il numero deve avere il primo trattino dopo il prefisso ❸, tre altri caratteri numerici ❹, poi un altro trattino ❽ e infine quattro ulteriori numeri ❻. Se l'esecuzione del programma riesce a superare tutti i controlli, restituisce `True` ⽾.

Se si chiama `isPhoneNumber()` con l'argomento '415-555-4242' si avrà di ritorno `True`. Se la si chiama con 'Moshi moshi' restituirà `False`; fallisce subito il primo test, perché 'Moshi moshi' non è una stringa lunga 12 caratteri.

Per identificare questo schema di testo in una stringa più lunga dovete aggiungere altro codice ancora. Sostituite le ultime quattro chiamate alla funzione `print()` in `isPhoneNumber.py` con queste righe:

```
message = 'Chiamami al 415-555-1011 domani. 415-555-9999 è il mio ufficio.'
for i in range(len(message)):
    ❶     chunk = message[i:i+12]
    ❷     if isPhoneNumber(chunk):
            print('Numero telefonico trovato: ' + chunk)
    print('Fatto')
```

Quando si esegue il programma, l'output è di questo tipo:

```
Numero telefonico trovato: 415-555-1011
Numero telefonico trovato: 415-555-9999
Fatto
```

A ogni iterazione del ciclo `for`, alla variabile `chunk` viene assegnato un nuovo blocco di 12 caratteri estratto da `message` ❶. Per esempio, alla prima iterazione `i` è 0 e a `chunk` viene assegnata `message[0:12]` (cioè la stringa 'Chiamami al'). All'iterazione successiva, `i` è 1 e a `chunk` viene assegnata `message[1:13]` (la stringa 'hiamami al 4').

Si passa `chunk` a `isPhoneNumber()` per stabilire se corrisponde allo schema di un numero telefonico ❷ e, nel caso, si stampa quella stringa.

Si continua a ciclare su `message` e a un certo punto i 12 caratteri in `chunk` saranno un numero telefonico. Il ciclo percorre tutta la stringa, verificando ciascun blocco di 12 caratteri e stampando ogni `chunk` che soddisfa `isPhoneNumber()`. Quando è stata esplorata tutta `message`, viene stampato `Fatto`.

La stringa in `message` in questo esempio è molto breve, ma potrebbe essere lunga anche milioni di caratteri e il programma verrebbe eseguito comunque in meno di un secondo. Un programma simile che cercasse i numeri telefonici utilizzando le espressioni regolari verrebbe a sua volta eseguito in meno di un secondo, ma le espressioni regolari rendono molto più veloce la scrittura di programmi di questo tipo.

## Trovare schemi di testo con le espressioni regolari

Il precedente programma per la ricerca di numeri telefonici funziona ma usa una gran quantità di codice per fare una cosa molto limitata: la funzione `isPhoneNumber()` è lunga 17 righe ma può trovare solo uno schema di numero telefonico. Che dire di un numero telefonico formattato come 415.555.4242 o (415 555-4242)? E se il numero telefonico avesse anche un interno, come 415-555-4242 x99? La funzione `isPhoneNumber()` non li riconoscerebbe. Potreste aggiungere altro codice per tener conto di questi schemi ulteriori, ma esiste un metodo più semplice.

Le **espressioni regolari** (indicate anche come *regex*, per brevità) sono descrizioni di uno **schemma di testo**. Per esempio `\d` in un'espressione regolare sta per una cifra (*digit*, in inglese), cioè qualsiasi singolo numerale fra 0 e 9. Python usa l'espressione `\d\d\d-\d\d\d-\d\d\d\d` per cercare lo stesso testo che cercava la precedente funzione `isPhoneNumber()`: una stringa di tre numeri, un trattino, tre numeri, un altro trattino, quattro numeri. Qualsiasi altra stringa non soddisfarebbe l'espressione `\d\d\d-\d\d\d-\d\d\d\d`. Le espressioni regolari però possono essere molto più complesse. Per esempio, se si aggiunge un `3` fra parentesi graffe `(\{3\})` dopo uno schema è come se si dicesse "Cerca questo schema ripetuto tre volte". Così anche l'espressione regolare, leggermente più breve della precedente, `\d\{3\}-\d\{3\}.\d\{4\}`

identifica le corrispondenze con il formato corretto dei numeri telefonici.

## Creare oggetti regex

Tutte le **funzioni** di Python per le **espressioni regolari** si trovano nel modulo `re`. Per importare questo modulo inserite quanto segue nella shell interattiva:

```
>>> import re
```

### NOTA

La maggior parte degli esempi che seguono, in questo capitolo, richiede il modulo `re`, perciò ricordate di importarlo all'inizio di ogni script che scrivete o ogni volta che riavviate IDLE. In caso contrario, vi ritroverete con un messaggio di errore `NameError: name 're' is not defined`.

Se si passa un valore stringa che rappresenta un'espressione regolare a `re.compile()`, si ottiene di ritorno un **oggetto schema Regex** (si dice anche “oggetto Regex”).

Per creare un oggetto Regex che corrisponda allo schema del numero telefonico, inserite quanto segue nella shell interattiva. (Ricordate che `\d` significa “una cifra” e che `\d\d\d-\d\d\d-\d\d\d` è l'espressione regolare per lo schema di un numero telefonico corretto.)

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d')
```

Ora la variabile `phonenumRegex` contiene un oggetto Regex.

### Passaggio di stringhe grezze a `re.compile()`

Ricordate che i caratteri escape in Python usano la barra retroversa (`\`). Il valore stringa `'\n'` rappresenta un singolo carattere di a capo, non una barra retroversa seguita da una *n* minuscola. Dovete inserire il carattere escape `\\\` per stampare una barra retroversa singola. Così `'\\n'` è la stringa che rappresenta una barra retroversa seguita da una lettera *n* minuscola. Se però mettete una `r` prima dell'apice di inizio del valore stringa, potete dire che la stringa è una stringa grezza, che non contiene caratteri escape.

Poiché le espressioni regolari usano spesso le barre retroverse, è comodo passare alla funzione `re.compile()` stringhe grezze anziché scrivere tante barre retroverse in più. È più facile scrivere `r'\d\d\d-\d\d\d-\d\d\d'` che non `'\\d\\d\\d-\\d\\d\\d-\\d\\d\\d'`.

## Corrispondenze con oggetti Regex

Il metodo `search()` di un oggetto Regex cerca ogni corrispondenza dell'espressione regolare nella stringa che gli viene passata come argomento. Il metodo `search()` restituisce `None` se lo schema dell'espressione non viene trovato nella stringa; se invece viene trovato, restituisce un oggetto `Match`. Gli oggetti `Match` hanno un metodo `group()` che restituisce l'effettivo testo identificato nella stringa su cui è effettuata la ricerca. (Parleremo dei gruppi fra breve.) Per esempio, inserite nella shell interattiva quanto segue:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d-')
>>> mo = phoneNumRegex.search('Il mio numero è 415-555-4242.')
>>> print('Numero telefonico trovato: ' + mo.group())
Numero telefonico trovato: 415-555-4242
```

Il nome di variabile `mo` è solo un nome generico da usare per gli oggetti Match. Questo esempio può sembrare complicato a prima vista, ma è molto più breve del precedente programma `isPhoneNumber.py`, e svolge lo stesso compito.

Qui, passiamo a `re.compile()` lo schema desiderato e memorizziamo in `phoneNumRegex` l'oggetto Regex risultante. Poi chiamiamo `search()` su `phoneNumRegex` e passiamo a `search()` la stringa in cui vogliamo cercare le eventuali corrispondenze. Il risultato della ricerca viene memorizzato nella variabile `mo`. In questo esempio, sappiamo che il nostro schema verrà trovato nella stringa, perciò sappiamo anche che verrà restituito un oggetto Match. Sapendo che `mo` contiene un oggetto Match e non il valore nullo `None`, possiamo chiamare `group()` su `mo` per restituire la corrispondenza. Scrivendo `mo.group()` all'interno dell'enunciato di stampa verrà visualizzata la corrispondenza completa, 415-555-4242.

## Riepilogo delle corrispondenze con espressioni regolari

L'uso delle espressioni regolari in Python si articola in vari passi, ma ciascun passo è piuttosto semplice.

1. Si importa il modulo per le espressioni regolari con `import re`.
2. Si crea un oggetto Regex con la funzione `re.compile()`. (Ricordate di usare una stringa grezza.)
3. Si passa la stringa su cui si vuole effettuare la ricerca al metodo `search()` dell'oggetto Regex. Verrà restituito un oggetto Match.
4. Si chiama il metodo `group()` dell'oggetto Match per avere di ritorno una stringa con il testo corrispondente identificato.

### NOTA

Vi invito a inserire il codice di esempio nella shell interattiva, ma dovreste utilizzare anche i tester web delle espressioni regolari, che possono mostrarvi esattamente le corrispondenze che un'espressione regolare trova all'interno di un brano di testo che potete inserire. Vi consiglio il tester che si può trovare all'indirizzo <http://regexpal.com>.

## Ancora corrispondenze con le espressioni regolari

Ora che conoscete i passi fondamentali per creare e trovare oggetti espressioni regolari in Python, siete pronti per mettere alla prova qualcuna delle loro capacità più potenti di ricerca di corrispondenze mediante schemi.

## Raggruppamento con le parentesi

Supponiamo che vogliate separare il prefisso dal resto del numero telefonico. Aggiungendo delle parentesi creerete dei gruppi nell'espressione regolare: `(\d\d\d)-(\d\d\d-\d\d\d\d)`. Poi potete usare il metodo `group()` dell'oggetto Match per ricavare il testo corrispondente da un solo gruppo.

Il primo insieme di parentesi in una stringa regex costituirà il gruppo 1; il secondo, il gruppo 2. Passando l'intero 1 o 2 al metodo `group()`, potrete estrarre parti diverse del testo corrispondente. Se passate al metodo `group()` uno 0 oppure nulla, avrete di ritorno tutto il testo corrispondente.

Inserite quanto segue nella shell interattiva:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('Il mio numero è 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

Se volete recuperare tutti i gruppi in una volta sola, usate il metodo `groups()` – notate che il nome ha una *s* finale, a indicare il plurale.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Poiché `mo.groups()` restituisce una tupla con più valori, potete utilizzare il trucco dell’assegnazione multipla per assegnare ciascun valore a una variabile diversa, come nel caso della precedente riga `areaCode, mainNumber = mo.groups()`.

Le parentesi hanno un significato speciale nelle espressioni regolari, ma che cosa si fa se si vuole cercare nel testo una **stringa che contiene parentesi**? Per esempio, magari i numeri telefonici che state cercando hanno il prefisso scritto fra parentesi. In questo caso, dovete effettuare l’escape dei caratteri ( e ) con una barra retroversa. Inserite quanto segue nella shell interattiva:

```
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('Il mio numero è (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

I caratteri escape \ ( e \) nella stringa grezza passata a `re.compile()` corrisponderanno a effettivi caratteri parentesi.

## Corrispondenza su più gruppi con il carattere pipe

Il carattere *pipe* () si può usare ovunque si voglia cerca **corrispondenze con una fra più espressioni**. Per esempio, l’espressione regolare r'Batman|Tina Fey' cercherà una corrispondenza con 'Batman' o con 'Tina Fey'.

Se nella stringa su cui si effettua la ricerca si trovano sia **Batman** che **Tina Fey**, verrà restituita come oggetto Match la prima occorrenza di testo corrispondente. Inserite quanto segue nella shell interattiva:

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman e Tina Fey.')
>>> mo1.group()
```

```
'Batman'  
>>> mo2 = heroRegex.search('Tina Fey e Batman.')  
>>> mo2.group()  
'Tina Fey'
```

## NOTA

Potete trovare tutte le occorrenze corrispondenti utilizzando il metodo `findall()`, di cui parleremo nel paragrafo “[Il metodo `findall\(\)`](#)” a pagina 150.

Potete usare il carattere pipe anche per cercare la corrispondenza con uno fra più modelli nella vostra espressione regolare. Per esempio, supponiamo che vogliate trovare una fra le stringhe 'Batman', 'Batmobile', 'Batcopter' e 'Batbat'. Poiché tutte queste stringhe iniziano con Bat, sarebbe bello poter specificare quel prefisso una volta sola. Lo si può fare, in effetti, con le parentesi. Inserite quanto segue nella shell interattiva:

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')  
>>> mo = batRegex.search('La Batmobile ha perso una ruota')  
>>> mo.group()  
'Batmobile'  
>>> mo.group(1)  
'mobile'
```

La chiamata al metodo `mo.group()` restituisce il testo completo individuato, 'Batmobile', mentre `mo.group(1)` restituisce solo quella parte del testo individuato che si trova all'interno del primo gruppo di parentesi, cioè 'mobile'. Utilizzando il carattere pipe e le parentesi per raggruppare, si possono specificare facilmente più schemi alternativi da cercare con le espressioni regolari.

Se dovete cercare un carattere pipe, ricordatevi di scriverlo come carattere escape, con la barra retroversa: `\|`.

## Corrispondenza facoltativa con il punto di domanda

A volte vi sono schemi che si vogliono **cercare solo facoltativamente**: in altre parole, l'espressione regolare deve trovare una corrispondenza, indipendentemente dal fatto che quel pezzo di testo sia presente o meno. Il carattere ? indica che il gruppo che lo precede è una parte facoltativa dello schema.

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> batRegex = re.compile(r'Bat(wo)?man')  
>>> mo1 = batRegex.search('The Adventures of Batman')  
>>> mo1.group()  
'Batman'  
  
>>> mo2 = batRegex.search('The Adventures of Batwoman')  
>>> mo2.group()  
'Batwoman'
```

La parte (wo)? dell'espressione regolare significa che lo schema **wo** è un gruppo facoltativo. L'espressione individuerà testo che ha al suo interno zero o una istanza di quel gruppo di caratteri. Per questo l'espressione individua sia 'Batwoman', sia 'Batman'.

Utilizzando l'esempio precedente del numero telefonico, potete scrivere un'espressione regolare che cerchi numeri telefonici con o senza un prefisso.

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d')
>>> mo1 = phoneRegex.search('Il mio numero è 415-555-4242')
>>> mo1.group()
'415-555-4242'

>>> mo2 = phoneRegex.search('Il mio numero è 555-4242')
>>> mo2.group()
'555-4242'
```

Potete pensare che il ? dica: "Cerca una stringa che contiene zero o una occorrenza del gruppo di caratteri che precede questo punto di domanda".

Se dovete **cercare un carattere punto di domanda**, trasformatelo in un carattere escape con la barra retroversa, \?.

## Trovare zero o più occorrenze con l'asterisco

L'asterisco \* significa "zero o più volte": il gruppo che lo precede può presentarsi un **numero qualsiasi di volte** nel testo. Può essere del tutto assente oppure ripetuto varie volte. Prendiamo ancora l'esempio di Batman.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

Per 'Batman' la parte (wo)\* dell'espressione regolare è soddisfatta con zero istanze di wo nella stringa; nel caso di 'Batwoman' con una istanza di wo; per 'Batwowowowoman' con quattro istanze di wo.

Se dovete **cercare un carattere asterisco**, trasformatelo in un carattere escape con la barra retroversa: \\*.

## Una o più corrispondenze con il segno "più"

Mentre l'asterisco significa "zero o più", il segno più (+) significa "**una o più**". A differenza dell'asterisco, che non richiede che il gruppo compaia nella stringa, il gruppo che precede un segno

più deve occorrere almeno una volta. Non è facoltativo. Inserite quanto segue nella shell interattiva, e fate il confronto con le espressioni regolari contenenti l'asterisco del paragrafo precedente:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'
>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

L'espressione regolare `Bat(wo)+man` non troverà la stringa 'The Adventures of Batman' perché il segno più richiede che sia presente almeno una volta il gruppo `wo`.

Se dovete **cercare un carattere “più”**, trasformatelo in un carattere escape con la barra retroversa: `\+`.

## Trovare un numero specifico di ripetizioni con le parentesi graffe

Se volete trovare un gruppo di caratteri che **si ripete un numero specifico di volte**, fate seguite al gruppo, nell'espressione regolare, un numero fra parentesi graffe. Per esempio, l'espressione regolare `(Ha){3}` troverà la stringa 'HaHaHa', ma non 'HaHa', perché in quest'ultima il gruppo `(Ha)` è ripetuto solo due volte.

Anziché un numero, potete **specificare** anche **un intervallo**, scrivendo un minimo, una virgola e un massimo fra le parentesi graffe. Per esempio, l'espressione regolare `(Ha){3,5}` troverà 'HaHaHa', 'HaHaHaHa' e 'HaHaHaHaHa'.

Se omettete il primo o il secondo numero fra parentesi graffe, il minimo o, rispettivamente, il massimo, non avranno limiti. Per esempio, `(Ha){3}` troverà tre o più istanze del gruppo `(Ha)`, mentre `(Ha){,5}` troverà da zero a cinque istanze. Le parentesi graffe possono contribuire a rendere più concise le vostre espressioni regolari. Per esempio, queste due espressioni regolari cercano gli stessi schemi:

```
(Ha){3}
(Ha)(Ha)(Ha)
```

E anche queste due espressioni regolari cercano gli stessi schemi:

```
(Ha){3,5}
((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))
```

Inserite quanto segue nella shell interattiva:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'
>>> mo2 = haRegex.search('Ha')
```

```
>>> mo2 == None
```

```
True
```

Qui, `(Ha){3}` identifica 'HaHaHa', ma non 'Ha'. Per questo, nel secondo caso `search()` restituisce `None`.

## Corrispondenze avide e non

`(Ha){3,5}` può individuare tre, quattro o cinque istanze di `Ha` nella stringa 'HaHaHaHaHa', perciò forse vi sarete chiesti perché la chiamata di `group()` nell'esempio precedente restituisca 'HaHaHaHaHa' anziché una delle possibilità più brevi. In fin dei conti, anche 'HaHaHa' e 'HaHaHaHa' sono corrispondenze valide per l'espressione `(Ha){3,5}`.

Le espressioni regolari di Python per default sono **avide**, il che significa che in situazioni ambigue privilegiano **la stringa più lunga possibile**. La versione *non avida* delle parentesi graffe, che sceglie la stringa più breve possibile, ha la parentesi graffa di chiusura seguita da un punto di domanda. Inserite quanto segue nella shell interattiva e notate la differenza fra la forma avida e quella non avida delle parentesi graffe, benché la ricerca riguardi la stessa stringa:

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'
>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?)')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Notate che il **punto di domanda** può avere **due significati** nelle espressioni regolari: dichiarare una corrispondenza non avida o evidenziare un gruppo facoltativo. Questi due significati non hanno alcuna relazione fra loro.

## Il metodo `findall()`

Oltre al metodo `search()`, gli oggetti Regex hanno un metodo `findall()`. Mentre `search()` restituisce un oggetto Match con il primo testo corrispondente nella stringa su cui si effettua la ricerca, `findall()` restituisce le strighe di **tutte le corrispondenze** nella stringa. Per vedere come `search()` restituisca un oggetto Match solo per la prima occorrenza del testo corrispondente, inserite quanto segue nella shell interattiva:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Ufficio: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

Invece `findall()` non restituirà un oggetto Match ma una lista di stringhe – *purché non esistano gruppi nell'espressione regolare*. Ciascuna stringa nella lista è una parte del testo su cui è stata effettuata la ricerca che soddisfa lo schema dell'espressione regolare. Inserite quanto segue nella shell interattiva:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
```

```
>>> phoneNumRegex.findall('Cell: 415-555-9999 Ufficio: 212-555-0000')
['415-555-9999', '212-555-0000']
```

Se invece *ci sono gruppi* nell'espressione regolare, `findall()` restituisce una **lista di tuple**. Ciascuna tupla rappresenta una corrispondenza individuata, e i suoi elementi sono le stringhe corrispondenti per ciascun gruppo dell'espressione regolare. Per vedere `findall()` in azione, inserite quanto segue nella shell interattiva (notate che l'espressione regolare ora ha dei gruppi fra parentesi):

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Ufficio: 212-555-0000')
[('415', '555', '1122'), ('212', '555', '0000')]
```

Per riepilogare quello che il metodo `findall()` restituisce, ricordate quanto segue.

1. Quando viene chiamato su una espressione regolare senza gruppi, come `\d\d\d-\d\d\d-\d\d\d\d`, il metodo `findall()` restituisce una lista di stringhe corrispondenti, come `['415-555-9999', '212-555-0000']`.
2. Quando è chiamato su un'espressione regolare contenente gruppi, per esempio `(\d\d\d)-(\d\d\d)-(\d\d\d\d)`, il metodo `findall()` restituisce una lista di tuple di stringhe (una stringa per ciascun gruppo), per esempio `[('415', '555', '1122'), ('212', '555', '0000')]`.

## Classi di caratteri

Nel precedente esempio di espressione regolare per il numero telefonico, avete visto che `\d` può stare per qualsiasi cifra. In altre parole, `\d` è una forma stenografica dell'espressione regolare `(0|1|2|3|4|5|6|7|8|9)`. Esistono molte di queste **classi di caratteri**, come è riportato nella [Tabella 7.1](#).

**Tabella 7.1** - Codici stenografici per classi di caratteri.

Classe di caratteri	Rappresenta
<code>\d</code>	Qualsiasi cifra, da 0 a 9
<code>\D</code>	Qualsiasi carattere che <i>non sia</i> una cifra
<code>\w</code>	Qualsiasi lettera, cifra o il carattere underscore. (Potete pensare che individui qualsiasi carattere che può comparire in una parola, <i>word</i> in inglese.)
<code>\W</code>	Qualsiasi carattere che <i>non sia</i> una lettera, una cifra o il carattere underscore.
<code>\s</code>	Qualsiasi carattere spazio, tabulazione o a capo. (Potete pensare che individui qualsiasi carattere "spazio".)
<code>\S</code>	Qualsiasi carattere che <i>non sia</i> uno spazio, una tabulazione o un a capo.

Le classi di caratteri sono comode per rendere concise le espressioni regolari. La classe di caratteri `[0-5]` individua solo i numeri da 0 a 5, per esempio, ed è una forma molto più breve rispetto a `(0|1|2|3|4|5)`. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
```

['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6  
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']

L'espressione regolare `\d+\s\w+` individua qualsiasi testo che abbia una o più cifre numeriche (`\d+`) seguite da un carattere spazio (`\s`), seguito da uno o più caratteri che siano lettere, cifre o underscore (`\w+`). Il metodo `findall()` restituisce in una lista tutte le stringhe corrispondenti allo schema dell'espressione regolare.

## Creare le proprie classi di caratteri

Si presentano occasioni in cui si vorrebbe effettuare una ricerca su un gruppo di caratteri, ma le classi predefinite (`\d`, `\w`, `\s` ecc.) sono troppo ampie. In quei casi, è possibile **definire classi** di caratteri **personalizzate**, utilizzando le parentesi quadre. Per esempio, la classe di caratteri `[aeiouAEIOU]` corrisponderà a qualsiasi vocale, maiuscola o minuscola. Inserite quanto segue nella shell interattiva:

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

Potete anche includere intervalli di lettere o numeri utilizzando un trattino. Per esempio la classe di caratteri `[a-zA-Z0-9]` comprende tutte le lettere minuscole e maiuscole e tutte le cifre.

Notate che, all'interno delle parentesi quadre, i normali simboli delle espressioni regolari non vengono interpretati come tali, e questo significa che non si deve effettuare l'escape di caratteri come `,`, `*`, `?` o `()` facendoli precedere da una barra retroversa. Per esempio, la classe di caratteri `[0-5.]` corrisponde alle cifre da 0 a 5 e al punto. Non bisogna scriverla nella forma `[0-5\.]`.

Inserendo un carattere accento circonflesso (`\`) subito dopo la parentesi quadra di apertura della classe di caratteri si può costruire una classe di caratteri negativa, che comprenderà tutti i caratteri che non sono nella classe di caratteri. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> consonantRegex = re.compile(r'[^\aeiouAEIOU]')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', 't', 's', 'b', 'b', 'y', 'f', 'd', ' ', '
', 'B', 'B', 'Y', 'F', 'D', '']
```

Ora, anziché individuare tutte le vocali, sono stati individuati tutti i caratteri che non sono vocali.

## I caratteri accento circonflesso e dollaro

Si può usare il simbolo dell'accento circonflesso anche all'inizio di una espressione regolare per indicare che la **corrispondenza** cercata deve presentarsi **all'inizio del testo**. Analogamente, si può inserire un segno di dollaro (\$) alla fine dell'espressione regolare per indicare che la stringa deve **finire con lo schema** dell'espressione regolare. Si possono anche usare i simboli `^` e `$` insieme, per indicare che **tutta la stringa deve corrispondere** all'espressione regolare – non basta, in altre parole, che la corrispondenza avvenga con qualche sottoinsieme della stringa.

Per esempio, l'espressione regolare `r'^Hello'` trova le stringhe che iniziano con 'Hello'. Inserite quanto segue nella shell interattiva:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

L'espressione regolare `r'\d$'` trova stringhe che terminano con un carattere numerico fra 0 e 9:

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<_sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

L'espressione regolare `r'^\d+$'` trova stringhe che iniziano e terminano con uno o più caratteri numerici:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12 34567890') == None
True
```

Le ultime due chiamate di `search()` nel precedente esempio nella shell interattiva dimostrano come l'intera stringa debba corrispondere all'espressione regolare se vengono usati `^` e `$`. Non dimenticate che l'accento circonflesso viene prima, il segno di dollaro dopo (capita di confondersi).

## Il carattere jolly

Il **carattere punto** `(.)` in una espressione regolare è un **jolly** e corrisponde a qualsiasi carattere che non sia un a capo. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Ricordate che il carattere punto corrisponde a un solo carattere: per questo nell'esempio precedente viene restituito solo `lat`, anche se la parola completa è `flat`. Per **trovare un punto**, il punto deve essere un carattere escape, preceduto una barra retroversa: `\.`.

## Trovare tutto con punto-asterisco

A volte si vuole che la corrispondenza sia con **qualsiasi cosa**. Per esempio, supponiamo che vogliate individuare la stringa 'Nome:', seguita da qualsiasi testo, seguita da 'Cognome:', seguita a sua volta da qualsiasi cosa. Potete usare la combinazione **punto-asterisco** `(*)` al posto di "qualsiasi cosa". Ricordate che il punto significa "qualsiasi singolo carattere, tranne l'a capo" e che l'asterisco significa "zero o più occorrenze del carattere precedente".

Inerite quanto segue nella shell interattiva:

```
>>> nameRegex = re.compile(r'Nome: (.*) Cognome: (.*)')
>>> mo = nameRegex.search('Nome: Al Cognome: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

La combinazione punto-asterisco usa la modalità *avid*: cercherà sempre di individuare il testo più lungo possibile. Per individuare tutto il testo in modo non avido, usate il punto, l’asterisco e il punto di domanda (*\*?*). Come nel caso delle parentesi graffe, il punto di domanda dice a Python di cercare una corrispondenza in modo non avido.

Inserite quanto segue nella shell interattiva, per vedere la differenza fra la versione avida e quella non avida:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner>')
>>> mo.group()
'<To serve man>'

>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner>')
>>> mo.group()
'<To serve man> for dinner.>'
```

Entrambe le espressioni regolari si traducono in “Cerca una parentesi angolare di apertura, seguita da qualsiasi cosa, seguita da una parentesi angolare di chiusura”, ma la stringa ‘<To serve man> for dinner.>’ ha due possibili corrispondenze per la parentesi angolare di chiusura. Nella versione avida dell’espressione regolare, Python trova la stringa più lunga possibile: ‘<To serve man> for dinner.>’.

## Trovare gli a capo con il carattere punto

La coppia punto-asterisco corrisponde a qualsiasi cosa, tranne un **a capo**. Passando a `re.compile()` come secondo argomento `re.DOTALL`, si può fare in modo che il carattere punto corrisponda a *tutti* i caratteri, compreso quello di a capo.

Inserite quanto segue nella shell interattiva:

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

L’espressione regolare `noNewlineRegex`, dove `re.DOTALL` non viene passata alla chiamata `re.compile` che l’ha creata, troverà qualsiasi cosa, ma solo fino al primo carattere di a capo, mentre `newlineRegex`, dove `re.DOTALL` viene passata a `re.compile`, trova tutto. Per questo la chiamata a `newlineRegex.search()` restituisce tutta la stringa, compresi i caratteri di a capo.

# Riepilogo dei simboli delle espressioni regolari

In questo capitolo abbiamo parlato di molte notazioni, perciò ecco un breve riepilogo di tutto quello che avete appreso.

- ? indica zero o una occorrenza del gruppo precedente.
- \* indica zero o più occorrenze del gruppo precedente.
- + indica una o più occorrenze del gruppo precedente.
- {n} indica esattamente  $n$  occorrenze del gruppo precedente.
- {n,} indica  $n$  o più occorrenze del gruppo precedente.
- {,m} indica da 0 a  $m$  occorrenze del gruppo precedente.
- {n,m} indica almeno  $n$  e non più di  $m$  occorrenze del gruppo precedente.
- ^spam significa che la stringa deve iniziare con *spam*.
- spam\$ significa che la stringa deve finire con *spam*.
- il punto (.) indica qualsiasi carattere, tranne i caratteri di a capo.
- \d, \w, \s indicano rispettivamente una cifra, un carattere non numerico e un carattere spazio.
- \D, \W, \S indicano qualsiasi cosa tranne, rispettivamente, una cifra, un carattere non numerico e un carattere spazio.
- [abc] indica qualsiasi carattere fra quelli fra parentesi (per esempio *a*, *b* o *c*).
- [^abc] indica qualsiasi carattere che non sia fra quelli fra parentesi.

## Corrispondenza case-insensitive

Normalmente, le espressioni regolari cercano una corrispondenza con l'esatta combinazione di **maiuscole** e **minuscole** specificata. Per esempio, le espressioni seguenti individuano stringhe del tutto diverse:

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

A volte però interessa solo trovare le lettere, non importa se siano maiuscole o minuscole. Per rendere *case-insensitive*, cioè insensibile alla differenza fra maiuscole e minuscole, un'espressione regolare, si può passare `re.IGNORECASE` o `re.I` come secondo argomento a `re.compile()`. Inserite quanto segue nella shell interattiva:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('AI, why does your programming book talk about robocop so much?').group()
'robocop'
```

## Sostituire stringhe con il metodo `sub()`

Le espressioni regolari possono non solo trovare schemi di testo ma anche sostituire a quegli schemi

**nuovo testo.** Al metodo `sub()` per gli oggetti Regex si passano due argomenti. Il primo è una stringa che andrà a sostituire qualsiasi occorrenza trovata, il secondo è la stringa per l'espressione regolare. Il metodo `sub()` restituisce una stringa in cui sono state applicate le sostituzioni.

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

A volte si ha bisogno di usare il testo cercato come parte della sostituzione. Nel primo argomento di `sub()` si può scrivere `\1`, `\2`, `\3` e così via, per indicare “Inserisci il testo del gruppo 1, 2, 3 e così via, nella sostituzione”.

Per esempio, supponiamo di voler censurare i nomi degli agenti segreti mostrando solo la prima lettera del loro nome. Potremmo usare l'espressione regolare `Agent (\w)\w*` e passare `r'\1****'` come primo argomento a `sub()`. `\1` in quella stringa sarà sostituito dal testo individuato dal gruppo 1, cioè il gruppo `(\w)` dell'espressione regolare.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob was a double agent.')
'A**** told C**** that E**** knew B**** was a double agent.'
```

## Gestire espressioni regolari complesse

Le espressioni regolari vanno bene se lo schema di testo che si deve individuare è semplice, ma individuare **schemi di testo complessi** può richiedere espressioni lunghe e un po' contorte. Si può ovviare in parte dicendo alla funzione `re.compile()` di ignorare spazi e commenti all'interno della stringa dell'espressione regolare. Questa modalità “loquace” (*verbose*, in inglese) può essere attivata passando la variabile `re.VERBOSE` come secondo argomento di `re.compile()`.

Ora, anziché un'espressione regolare difficile da leggere come questa:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?)?(\s|-\.)?\d{3}(\s|-\.)\d{4}(\s*(ext|x|ext.)\s*\d{2,5})?')'
```

si può distribuire l'espressione su più righe, inserendo anche dei commenti, così:

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?          # prefisso
    (\s|-\.)?                  # separatore
    \d{3}                      # prime 3 cifre
    (\s|-\.)                   # separatore
    \d{4}                      # ultime 4 cifre
    (\s*(ext|x|ext.)\s*\d{2,5})? # estensione (ext)
)''', re.VERBOSE)
```

Notate come l'esempio precedente usi la **sintassi dei triplici apici** ("""") per creare una **stringa multiriga** in modo da poter distribuire la definizione dell'espressione regolare su molte righe, rendendola molto più leggibile.

Le regole per i commenti all'interno della stringa di un'espressione regolare sono le stesse del normale codice Python: il simbolo `#` e tutto ciò che lo segue fino alla fine della riga sono ignorati. Inoltre, gli spazi aggiuntivi all'interno della stringa multiriga dell'espressione regolare non sono

considerati parte dello schema di testo da individuare. Questo permette di organizzare l'espressione regolare nel modo migliore per renderla il più facilmente leggibile.

## Combinare re.IGNORECASE, re.DOTALL e re.VERBOSE

E se si volesse usare `re.VERBOSE` per scrivere commenti in un'espressione regolare ma si volesse usare anche `re.IGNORECASE` per trascurare la differenza fra maiuscole e minuscole? Purtroppo, la funzione `re.compile()` accetta un solo valore come suo secondo argomento. Si può aggirare questa limitazione combinando le variabili `re.IGNORECASE`, `re.DOTALL` e `re.VERBOSE` utilizzando il carattere pipe (`|`), che in questo contesto è chiamato **operatore or su bit**.

Così, se volete un'espressione regolare non sensibile a maiuscole-minuscole e che includa gli a capo fra i caratteri individuati dal punto, dovete costruire la vostra chiamata a `re.compile()` in questo modo:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

Tutte le tre opzioni per il secondo argomento andranno inserite in questo modo:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

Questa sintassi è un po' antiquata e deriva dalle prime versioni di Python. I dettagli su come funzionano gli operatori su bit vanno al di là degli obiettivi di questo libro, ma trovate maggiori informazioni fra le risorse all'indirizzo <http://nostarch.com/automatestuff/>. Potete passare anche altre opzioni come secondo argomento: non sono molto utilizzate, ma potete scoprire di più sempre fra le risorse in rete.

## Progetto: estrarre numeri telefonici e indirizzi email

Supponiamo che abbiate il noioso compito di trovare ogni numero di telefono e ogni indirizzo email in una lunga pagina web o in un lungo documento. Se scorrete manualmente la pagina, potreste dover cercare a lungo, ma se avete un programma che esaminasse il testo negli Appunti alla ricerca di numeri telefonici e indirizzi email, potreste semplicemente premere Ctrl-A per selezionare tutto il testo, premere Ctrl-C per copiarlo negli Appunti e poi eseguire il programma. Potrebbe sostituire il testo negli Appunti con i soli numeri telefonici e indirizzi email che trova.

Quando si affronta un nuovo progetto, si può essere tentati di buttarsi subito a capofitto nella scrittura del codice, ma nella maggior parte dei casi è meglio fare un passo indietro e **analizzare il quadro** complessivo.

Consiglio di tracciare prima un piano di alto livello per individuare quello che il programma deve fare: non pensate ancora al codice effettivo, ve ne preoccuperete dopo. Per ora, limitatevi ai tratti più generali.

Per esempio, il vostro estrattore di numeri telefonici e indirizzi email deve:

- prendere il testo dagli Appunti;
- trovare nel testo tutti i numeri telefonici e gli indirizzi email;
- incollarli negli Appunti.

Ora potete cominciare a pensare come tutto questo si possa realizzare nel codice. Il codice dovrà fare queste cose:

- usare il modulo `pyperclip` per copiare e incollare stringhe;
- creare due regex, una per i numeri telefonici e l'altra per gli indirizzi email;
- trovare tutte le occorrenze, e non solo la prima, di entrambe le espressioni regolari;
- formattare bene le stringhe identificate mettendole in un'unica stringa da incollare;
- visualizzare qualche tipo di messaggio se non trova alcuna corrispondenza nel testo.

Questo elenco è come un itinerario per il progetto. Nello scrivere il codice, potete concentrarvi su ciascuno di questi passi separatamente. Ciascun passo è ben gestibile ed è espresso in termini di cose che già sapete come fare in Python.

## Passo 1: creare un'espressione regolare per i numeri telefonici

In primo luogo, dovete creare un'espressione regolare per cercare i numeri telefonici. Create un nuovo file nel file editor, inserite il codice seguente e salvate il file con il nome `phoneAndEmail.py`.

```
#! python3
# phoneAndEmail.py - Trova negli Appunti numeri telefonici e indirizzi email.

import pyperclip, re

phoneRegex = re.compile(r'''(
    (\d{3})|(\d{3}\))?
    (\s|-|\.)?
    (\d{3})
    (\s|-|\.)
    (\d{4})
    (\s*(ext|x|ext.)\s*(\d{2,5}))?
)''', re.VERBOSE)

# DA FARE: Creare espressione regolare per email.

# DA FARE: Trovare corrispondenze nel testo negli Appunti.

# DA FARE: Copiare i risultati negli appunti.
```

I commenti DA FARE sono solo lo scheletro del programma: verranno sostituiti a mano a mano che scriverete il codice effettivo.

Il numero telefonico inizia con un prefisso *facoltativo*, perciò il gruppo del prefisso è seguito da un punto di domanda. Poiché il prefisso può essere formato da tre sole cifre (cioè `\d{3}`) oppure da tre cifre entro parentesi (cioè, `\(\d{3}\)`), queste parti devono essere connesse con un pipe. Potete anche aggiungere il commento `# Prefisso` a questa parte della stringa multiriga dell'espressione regolare per ricordarvi a che cosa deve corrispondere `(\d{3})|(\d{3}\))?`.

Il carattere separatore del numero telefonico può essere uno spazio (`\s`), un trattino (-) o un punto (.), perciò anche queste parti devono essere connesse da pipe. Le altre parti dell'espressione regolare sono immediate: tre cifre, seguite da un altro separatore, seguite da quattro cifre. L'ultima parte è un'estensione facoltativa (un interno) costituito da un numero qualsiasi di spazi seguito da ext, x, o ext., seguito da un numero di cifre compreso fra due e cinque.

## Passo 2: creare un'espressione regolare per gli indirizzi email

Vi serve un'espressione regolare anche per identificare gli indirizzi email. Questa parte del programma sarà come questa:

```
#! python3
# phoneAndEmail.py - Trova negli Appunti numeri telefonici e indirizzi email.

import pyperclip, re

phoneRegex = re.compile(r'''(
--qui le altre righe--

# Crea espressione regolare per email.
emailRegex = re.compile(r'''(
❶ [a-zA-Z0-9._%+-]+          # username
❷ @                          # simbolo @
❸ [a-zA-Z0-9.-]+            # nome del dominio
(\.[a-zA-Z]{2,4})           # punto-qualcetosa
)'''', re.VERBOSE)

# DA FARE: Trovare corrispondenze nel testo negli Appunti.

# DA FARE: Copiare i risultati negli Appunti.
```

La parte relativa al nome utente (*username*) dell'indirizzo email ❶ è costituita da uno o più caratteri che possono essere qualsiasi combinazione di lettere maiuscole e minuscole, numeri, un punto, un underscore, un segno di percentuale, un segno più o un trattino. Potete inserirli tutti in una classe di caratteri: [a-zA-Z0-9.\_%+-].

Il dominio e il nome utente sono separati da un simbolo @ ❷. Il nome del dominio ❸ ha una classe di caratteri un po' più ristretta, a cui appartengono solo lettere, numeri, punti e trattini: [a-zA-Z0-9.-]. Infine c'è la parte “dot-com” (teoricamente, il *dominio di livello più alto*), che in realtà può essere formata da un punto seguito da qualsiasi cosa, per un numero di caratteri compreso fra due e quattro. Il formato degli indirizzi email deve rispettare regole complesse. Questa espressione regolare non catturerà tutti i possibili indirizzi validi, ma quasi tutti quelli più comuni.

### Passo 3: trovare tutte le corrispondenze nel testo negli Appunti

Ora che avete specificato le espressioni regolari per i numeri telefonici e gli indirizzi email, potete far fare al modulo `re` di Python il lavoro pesante, trovare negli Appunti tutte le corrispondenze.

La funzione `pyperclip.paste()` otterrà un valore stringa del testo presente negli Appunti, e il metodo `regex.findall()` restituirà una lista di tuple.

Ecco come dovete ampliare il vostro programma:

```

#! python3
# phoneAndEmail.py - Trova negli Appunti numeri telefonici e indirizzi email.

import pyperclip, re

phoneRegex = re.compile(r'''(
--qui le altre righe--

# Trova corrispondenze nel testo negli Appunti.
text = str(pyperclip.paste())
❶ matches = []
❷ for groups in phoneRegex.findall(text):
    phoneNum = '-'.join([groups[1], groups[3], groups[5]])
    if groups[8] != '':
        phoneNum += ' x' + groups[8]
    matches.append(phoneNum)
❸ for groups in emailRegex.findall(text):
    matches.append(groups[0])

# DA FARE: Copiare i risultati negli Appunti.

```

Vi è una tupla per ciascuna corrispondenza, e ogni tupla contiene stringhe per ciascun gruppo nell'espressione regolare. Ricordate che il gruppo 0 corrisponde all'espressione regolare completa, perciò il gruppo all'indice 0 della tupla è quello che vi interessa.

Come si può vedere in ❶, le corrispondenze vengono salvate in una variabile lista chiamata `matches`. All'inizio è una lista vuota e si parte con un paio di cicli `for`. Per gli indirizzi email, si accoda il gruppo 0 di ciascuna corrispondenza ❸. Per i numeri telefonici, non volete accodare semplicemente il gruppo 0. Mentre il programma identifica i numeri telefonici in vari formati, volete che il numero telefonico accodato sia in un ben preciso formato standard.

La variabile `phoneNum` contiene una stringa costruita a partire dai gruppi 1, 3, 5 e 8 del testo identificato ❷. (Questi gruppi sono il prefisso, le prime tre cifre, le ultime quattro cifre e l'interno.)

## Passo 4: unire le corrispondenze in una stringa per gli Appunti

Ora che avete indirizzi email e numeri telefonici come una lista di stringhe in `matches`, volete portarli negli Appunti. La funzione `pyperclip.copy()` prende solo un valore stringa, non una lista di stringhe, perciò dovete chiamare il metodo `join()` su `matches`. Perché sia più facile vedere che il programma sta lavorando, stampiamo sul terminale le corrispondenze identificate; se poi non vengono trovati né indirizzi email né numeri telefonici, il programma deve comunicarlo. Modificate il vostro programma in questo modo:

```
#! python3
# phoneAndEmail.py - Trova negli Appunti numeri telefonici e indirizzi email.

--righe omesse--
for groups in emailRegex.findall(text):
    matches.append(groups[0])

# Copia i risultati negli Appunti.
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copiato negli Appunti:')
    print('\n'.join(matches))
else:
    print('Non ho trovato né numeri telefonici né indirizzi email.')
```

## Esecuzione del programma

Per un esempio, aprete il browser web, andate alla pagina dei contatti di No Starch Press all'indirizzo <http://www.nostarch.com/contactus.htm>, premete Ctrl-A per selezionare tutto il testo della pagina e premete Ctrl-C per copiarlo negli Appunti. Quando eseguite questo programma, l'output sarà analogo a questo:

```
Copiato negli Appunti:
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
help@nostarch.com
```

## Idee per programmi simili

L'identificazione di schemi di testo (ed eventualmente la loro sostituzione con il metodo `sub()`) ha molte applicazioni possibili.

- Trovare URL di siti web che iniziano con `http://` o `https://`.
- Unificare date scritte in formati diversi (per esempio 14/3/2018, 14-03-2018, 14-3-2018, oppure, all'ingrese, 3/14/2018, 03/14/2018, 2018/3/14), scrivendole tutte in un formato standard.
- Eliminare informazioni delicate, come i numeri delle carte di credito.
- Trovare refusi comuni, come i doppi spazi fra parole, parole ripetute ripetute, o i molti punti esclamativi alla fine di una frase, che sono proprio irritanti!!!

## Riepilogo

Un computer può effettuare una **ricerca in un testo** molto rapidamente, ma bisogna dirgli con estrema precisione che cosa deve cercare. Le **espressioni regolari** consentono di specificare estattamente gli **schemi di caratteri** che si vogliono trovare: anche alcuni programmi di elaborazione testi e fogli di calcolo dispongono di funzioni di ricerca e sostituzione che permettono l'uso delle espressioni regolari.

Il modulo `re` che è fornito con Python permette di compilare oggetti Regex. Questi valori hanno vari metodi: `search()` per trovare una singola corrispondenza, `.findall()` per trovare tutte le occorrenze corrispondenti e `sub()` per la ricerca e sostituzione di testo.

Vi sono altri elementi della sintassi delle espressioni regolari che non sono trattati in questo capitolo. Potete trovare ulteriori informazioni nella documentazione ufficiale di Python all'indirizzo <http://docs.python.org/3/library/re.html>. Come risorsa può essere utile anche il tutorial <http://www.regular-expressions.info>.

Ora che avete fatto un po' di esperienza nella manipolazione e nell'identificazione delle stringhe, è venuto il momento di parlare dei file che si trovano sul disco fisso del computer, di come si possa leggerli e di come vi si possa scrivere.

## Domande di ripasso

1. Qual è la funzione che crea oggetti Regex?
2. Perché, nella creazione di oggetti Regex, si usano spesso stringhe grezze?
3. Che cosa restituisce il metodo `search()`?
4. Come si ottengono le stringhe effettive che corrispondono allo schema da un oggetto `Match`?
5. Nell'espressione regolare creata da `r'(\d\d\d)-(\d\d\d-\d\d\d\d\d)`, che cosa copre il gruppo 0? E il gruppo 1? Il gruppo 2?
6. Parentesi e punti hanno significati specifici nella sintassi delle espressioni regolari. Come potete specificare che volete un'espressione regolare che cerchi caratteri parentesi e punto?
7. Il metodo `.findall()` restituisce una lista di stringhe o una lista di tuple di stringhe. Da che cosa dipende che il risultato sia di un tipo o dell'altro?
8. Che cosa significa il carattere `|` nelle espressioni regolari?
9. Quali sono i due significati che può avere il carattere `?` nelle espressioni regolari?
10. Qual è la differenza fra i caratteri `+` e `*` nelle espressioni regolari?
11. Qual è la differenza fra `{3}` e `{3,5}` nelle espressioni regolari?
12. Che cosa significano le classi di caratteri `\d`, `\w`, `\s` nelle espressioni regolari?
13. Che cosa significano le classi di caratteri `\D`, `\W`, `\S` nelle espressioni regolari?
14. Come si può rendere *case-insensitive* un'espressione regolare?
15. A che cosa corrisponde normalmente il carattere `.` (punto)? A che cosa corrisponde, se si passa `re.DOTALL` come secondo argomento a `re.compile()`?
16. Qual è la differenza fra `.*` e `.*??`?
17. Qual è la sintassi a classi di caratteri per trovare tutti i numeri e le lettere minuscole?
18. Se `numRegex = re.compile(r'\d+')`, che cosa restituirà `numRegex.sub('X', '12 drummers, 11 pipers, five rings, 3 hens?')`?
19. Che cosa è possibile fare, se si passa `re.VERBOSE` come secondo argomento a `re.compile`?
20. Come scrivereste un'espressione regolare che identifichi numeri con la virgola come separatore ogni tre cifre? Deve identificare i seguenti:
  - '42'
  - '1,234'
  - '6,368,745'ma non:
  - '12,34,567' (che ha solo due cifre fra le virgole)
  - '1234' (che non ha virgole)
21. Come scrivereste un'espressione regolare che identifichi il nome completo di qualcuno il cui cognome è Nakamoto? Potete ipotizzare che il nome che lo precede sia sempre una parola che

inizia con una lettera maiuscola. L'espressione deve identificare:

- 'Satoshi Nakamoto'
- 'Alice Nakamoto'
- 'RoboCop Nakamoto'

ma non:

- 'satoshi Nakamoto' (il nome non ha l'iniziale maiuscola)
- 'Mr. Nakamoto' (la parola che precede il cognome ha un carattere che non è una lettera)
- 'Nakamoto' (non c'è il nome)
- 'Satoshi nakamoto' (il cognome non ha l'iniziale maiuscola).

22. Come scrivereste un'espressione regolare che identifichi una frase in cui la prima parola è *Alice*, *Bob* o *Carol*; la seconda parola è *mangia*, *coccola* o *lancia*, la terza parola è *mele*, *gatti* o *palline*; e la frase finisce con un punto? L'espressione regolare deve essere case-insensitive. Deve individuare:

- 'Alice mangia mele.'
- 'Bob coccola gatti.'
- 'Carol lancia palline.'
- 'Alice lancia Mele.'
- 'BOB MANGIA GATTI'

ma non:

- 'RoboCop mangia mele.'
- 'ALICE LANCIA PALLONI.'
- 'Carol mangia 7 gatti.'

## Un po' di pratica

Per esercitarvi, scrivete programmi che svolgano queste attività.

### Identificazione di password robuste

Scrivete una funzione che usi espressioni regolari per verificare che la stringa di pass-word che le viene passata è robusta. Una password è robusta se è lunga almeno otto caratteri, contiene caratteri sia minuscoli che maiuscoli e contiene almeno una cifra. Per valutare la robustezza, dovete verificare la stringa rispetto a più schemi di espressioni regolari.

### Versione a espressione regolare di strip()

Scrivete una funzione che prenda una stringa e faccia la stessa cosa del metodo stringa `strip()`. Se non viene passato alcun argomento oltre alla stringa su cui intervenire, deve eliminare gli spazi in più a inizio e fine stringa; altrimenti eliminerà dalla stringa i caratteri specificati nel secondo argomento della funzione.

---

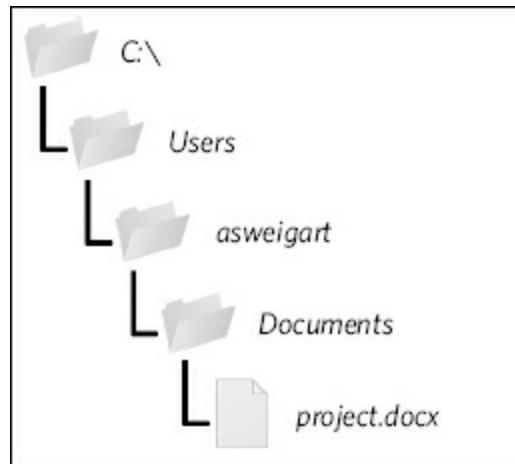
<sup>1</sup> Cory Doctorow, "Here's what ICT should really teach kids: how to do regular expressions", Guardian, 4 dicembre 2012, <http://www.theguardian.com/technology/2012/dec/04/ict-teach-kids-regular-expressions/>.

# Leggere e scrivere file

Le **variabili** sono un buon modo per memorizzare dati mentre il programma è in esecuzione, ma se volete che i **dati** persistano anche dopo la fine del programma, dovete salvarli in un **file**. Potete pensare i contenuti di un file come un **unico valore stringa**, potenzialmente delle dimensioni dei gigabyte. In questo capitolo, vedrete come usare Python per **creare, leggere e salvare file** sul disco fisso.

## File e percorsi di file

Un file ha due proprietà fondamentali: un **nome di file** (di solito scritto come una parola sola) e un **percorso** (*path*). Il percorso specifica la posizione del file nel computer. Per esempio, sul mio laptop con Windows 7 si trova un file il cui nome è *projects.docx*, nel percorso *C:\Users\asweigart\Documents*. La parte del nome del file dopo l'ultimo punto si chiama **estensione** o **suffisso** del file e dice di che *tipo* sia quel file. *project.docx* è un documento Word, mentre *Users*, *asweigart* e *Documents* si riferiscono tutti a *cartelle* (chiamate anche *directory*) Le cartelle possono contenere file e altre cartelle. Per esempio, *project.docx* è nella cartella *Documents*, che sta nella cartella *asweigart*, che si trova nella cartella *Users*. La [Figura 8.1](#) mostra questa organizzazione a cartelle.



**Figura 8.1** - Un file in una gerarchia di cartelle.

La parte *C:\* del percorso è la **cartella radice**, che contiene tutte le altre cartelle. In Windows, la cartella radice si chiama *C:\* ed è chiamata anche **unità** (o **drive**) *C:*. Sotto OS X e Linux, la cartella radice è */*. In questo libro, userò la cartella radice in stile Windows, cioè la indicherò come *C:\*. Se inserite gli esempi della shell interattiva su sistemi con OS X o Linux, scrivete invece */*.

Altri *volumi*, come una unità DVD o una chiavetta USB, appariranno in modo diverso sui diversi sistemi operativi. In Windows, si presentano come nuove unità radice identificate da una lettera, come *D:\* o *E:\*. In OS X, compaiono come nuove cartelle sotto la cartella */Volumes*. In Linux, sono nuove cartelle sotto la cartella */mnt* (“mount”). Notate inoltre che nomi di cartelle e nomi di file non fanno distinzione fra maiuscole e minuscole in Windows e OS X, mentre maiuscole e minuscole sono considerate diverse in Linux.

## Barra retroversa in Windows, normale in OS X e Linux

In Windows, i **percorsi** vengono scritti utilizzando le **barre retroverse** (*\*) come separatori fra i nomi di cartella. OS X e Linux, invece, usano la barra normale (*/*) come separatore nei percorsi. Se volete che i vostri programmi funzionino su tutti i sistemi operativi, dovrete scrivere i vostri script Python in modo che gestiscano entrambi i casi.

Per fortuna, questo è facile con la funzione `os.path.join()`. Se passate a questa funzione i valori stringa dei nomi dei file e delle cartelle nel vostro percorso, `os.path.join()` restituirà una stringa con un percorso di file che usa i separatori corretti. Inserite quanto segue nella shell interattiva:

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\bin\spam'
```

Esegui questi esempi nella shell interattiva in Windows, perciò `os.path.join('usr', 'bin', 'spam')` mi ha restituito `'usr\bin\spam'`. (Notate che le barre sono doppie, perché la barra retroversa deve diventare un **carattere escape** con un altro carattere barra retroversa). Se avessi chiamato questa funzione in OS X o in Linux, la stringa sarebbe stata `'usr/bin/spam'`.

La funzione `os.path.join()` è utile se dovete creare stringhe per nomi di file. Queste stringhe verranno passare a molte delle funzioni relative ai file di cui parleremo in questo capitolo. Per esempio, il codice seguente aggiunge alla fine del nome di una cartella i nomi di file contenuti in una lista:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
```

```
>>> for fileName in myFiles:  
    print(os.path.join('C:\\Users\\asweigart', fileName))  
C:\\Users\\asweigart\\accounts.txt  
C:\\Users\\asweigart\\details.csv  
C:\\Users\\asweigart\\invite.docx
```

## La directory corrente

Ogni programma che viene eseguito sul vostro computer ha una **directory di lavoro corrente** (per brevità, **cwd**, da *current working directory*). Qualsiasi nome di file o percorso che non inizi con la cartella radice si dà per scontato che si trovi nella directory di lavoro corrente. Potete ottenere la directory di lavoro corrente come valore stringa con la funzione `os.getcwd()` e potete modificarla con `os.chdir()`. Inserite quanto segue nella shell interattiva:

```
>>> import os  
>>> os.getcwd()  
'C:\\Python34'  
>>> os.chdir('C:\\Windows\\System32')  
>>> os.getcwd()  
'C:\\Windows\\System32'
```

Qui la directory di lavoro corrente è impostata a *C:\\Python34*, perciò il nome di file *project.docx* si riferisce a *C:\\Python34\\project.docx*. Se cambio la directory di lavoro corrente in *C:\\Windows*, *project.docx* è interpretato come *C:\\Windows\\project.docx*.

Python visualizzerà un errore se tentate di cambiare la cwd in una directory che non esiste:

```
>>> os.chdir('C:\\QuestaCartellaNonEsiste')  
Traceback (most recent call last):  
File "<pyshell#18>", line 1, in <module>  
os.chdir('C:\\ThisFolderDoesNotExist')  
FileNotFoundException: [WinError 2] The system cannot find the file specified:  
'C:\\QuestaCartellaNonEsiste'
```

### NOTA

“Cartella” è il nome più moderno per indicare una directory, ma notate che il termine standard è “directory di lavoro corrente” (o semplicemente “directory corrente”) e non “cartella di lavoro corrente”.

## Percorsi assoluti e relativi

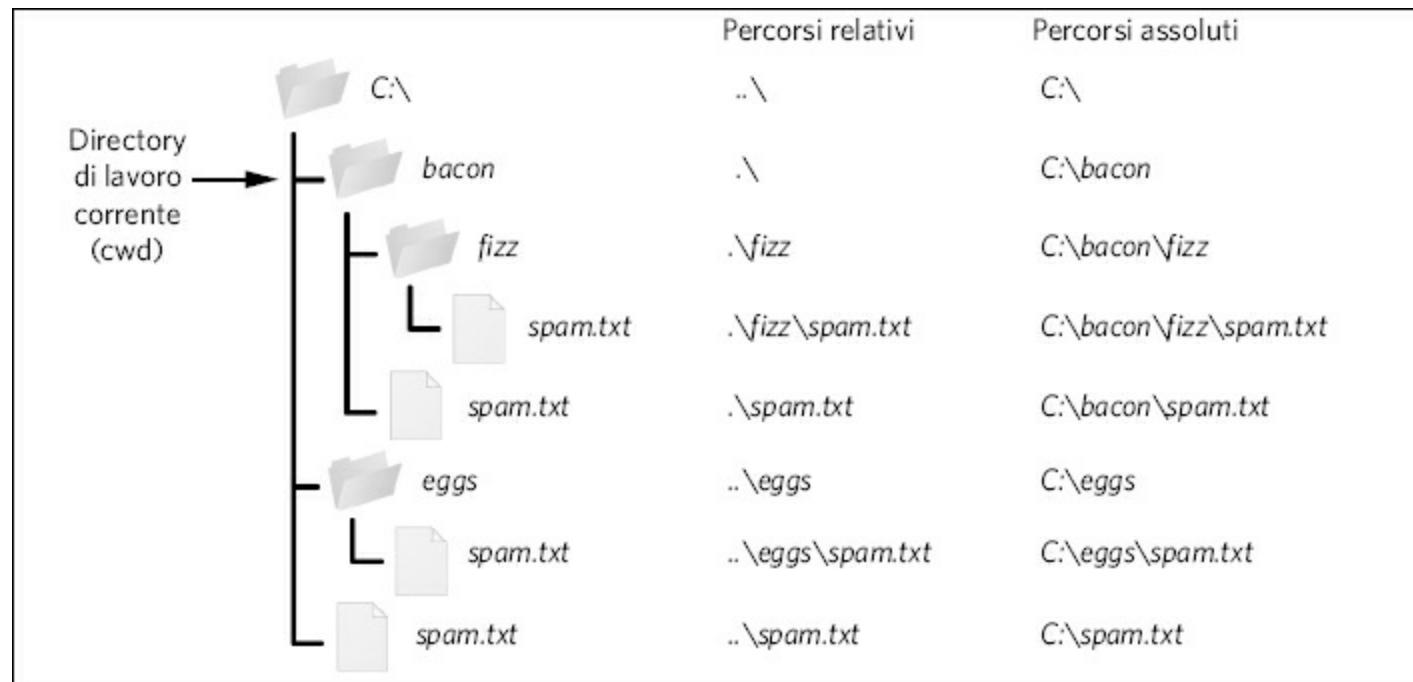
Esistono due modi per specificare un percorso di file.

- Un **percorso assoluto** inizia sempre con la cartella radice.
- Un **percorso relativo** è relativo alla directory di lavoro corrente del programma.

Esistono anche le **cartelle punto** (.) e **punto-punto** (...). Queste non sono però cartelle reali, ma nomi speciali utilizzabili in un percorso. Un singolo punto (o “dot”) al posto di un nome di cartella è una

forma abbreviata per dire “questa directory”. Due punti (o “dot-dot”) significa “la cartella genitrice”, ovvero la cartella immediatamente superiore (quella che contiene la cartella corrente).

La Figura 8.2 presenta alcuni esempi di cartelle e file. Quando la directory di lavoro corrente è `C:\bacon`, i percorsi relativi per le altre cartelle e gli altri file sono costruiti come nella figura.



**Figura 8.2** - I percorsi relativi per cartelle e file quando la directory di lavoro corrente è C:\bacon.

La combinazione di simboli `\.` all'inizio di un percorso relativo è facoltativa. Per esempio, `.\spam.txt` e `spam.txt` si riferiscono allo stesso file.

## Creare nuove cartelle con os.makedirs()

I vostri programmi possono creare **nuove cartelle** (directory) con la funzione `os.makedirs()`. Inserite quanto segue nella shell interattiva:

```
>>> import os  
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

Questo creerà non solo la cartella `C:\delicious` ma anche la cartella `walnut` all'interno di `C:\delicious` e una cartella `waffles` all'interno di `C:\delicious\walnut`. In altre parole, `os.makedirs()` crea anche tutte le cartelle intermedie necessarie perché esista il percorso completo. La [Figura 8.3](#) rappresenta questa gerarchia di cartelle.

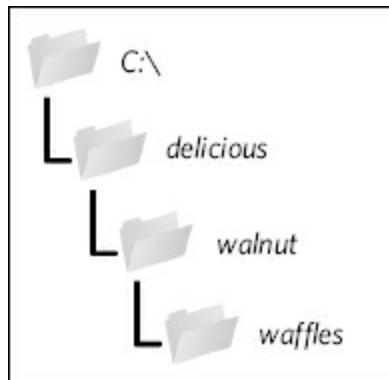


Figura 8.3 - Il risultato di os.makedirs('C:\\delicious\\walnut\\waffles').

## Il modulo `os.path`

Il modulo `os.path` contiene molte funzioni utili relative a nomi di file e percorsi di file. Per esempio, avete già utilizzato `os.path.join()` per costruire percorsi in modo che funzionino su qualsiasi sistema operativo. Poiché `os.path` è un modulo all'interno del modulo `os`, potete importarlo semplicemente eseguendo `import os`. Ogni volta che i vostri programmi devono lavorare con file, cartelle o percorsi di file, potete consultare i brevi esempi di questa sezione. La documentazione completa per il modulo `os.path` si trova sul sito web di Python, <http://docs.python.org/3/library/os.path.html>.

### NOTA

La maggior parte degli esempi in questa sezione richiede il modulo `os`, perciò ricordatevi sempre di importarlo all'inizio di tutti gli script che scrivete e ogni volta che riavviate IDLE, altrimenti vi si presenterà un messaggio d'errore `NameError: name 'os' is not defined`.

## Gestire percorsi assoluti e relativi

Il modulo `os.path` mette a disposizione **funzioni** che restituiscono il **percorso assoluto** di un percorso relativo e che controllano se un percorso dato è un percorso assoluto.

- La chiamata `os.path.abspath(percuso)` restituisce una stringa con il percorso assoluto dell'argomento. Questo è un modo semplice per convertire un percorso relativo in un percorso assoluto.
- La chiamata `os.path.isabs(percuso)` restituisce `True` se l'argomento è un percorso assoluto e `False` se è un percorso relativo.
- La chiamata `os.path.relpath(percuso, inizio)` restituisce una stringa con il percorso relativo dal percorso `inizio` al percorso `percuso`. Se non si indica il percorso `inizio`, viene utilizzato come percorso iniziale quello della directory di lavoro corrente.

Provate queste funzioni nella shell interattiva:

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
```

```
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Poiché quando è stata chiamata `os.path.abspath()` la directory corrente era `C:\Python34`, la cartella “punto” rappresenta il percorso assoluto ‘`C:\\Python34`’.

## NOTA

Con tutta probabilità il vostro sistema ha file e cartelle con nomi diversi dai miei, non potrete seguire tutti gli esempi di questo capitolo alla lettera. Cercate comunque di seguire il testo utilizzando nomi di file e di cartelle che esistono sul vostro disco fisso.

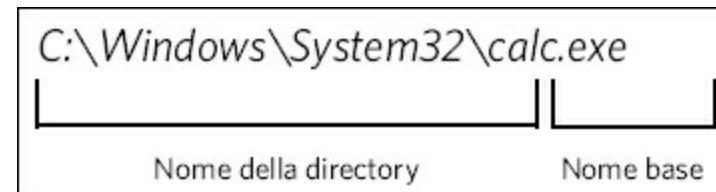
Inserite queste chiamate a `os.path.relpath()` nella shell interattiva:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
'C:\\Python34'
```

La chiamata `os.path.dirname(percors)` restituirà una stringa contenente tutto ciò che viene prima dell’ultima barra nell’argomento `percors`.

La chiamata `os.path.basename(percors)` restituirà una stringa con tutto quello che viene dopo l’ultima barra nell’argomento `percors`.

Il **nome della directory** (*dir name*) e il **nome base** (*base name*) di un percorso sono rappresentati nella [Figura 8.4](#).



**Figura 8.4** - Il nome base segue l’ultima barra in un percorso e coincide con il nome di file. Il nome di directory è tutto ciò che precede l’ultima barra.

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

Se avete bisogno del nome della directory e del nome base di un percorso insieme, potete semplicemente chiamare `os.path.split()` per avere un valore tupla con queste due stringhe, per esempio

così:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'  
>>> os.path.split(calcFilePath)  
('C:\\Windows\\System32', 'calc.exe')
```

Notate che potreste creare la stessa tupla chiamando `os.path.dirname()` e `os.path.basename()` e poi mettendo i loro valori di ritorno in una tupla:

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))  
('C:\\Windows\\System32', 'calc.exe')
```

`os.path.split()` però è una bella scorciatoia, se vi servono entrambi i valori.

Notate inoltre che `os.path.split()` non prende un percorso di file per restituire una lista di stringhe di ciascuna cartella. Per avere questo risultato, utilizzate il metodo stringa `split()` e dividete la stringa in `os.sep`. Ricordate, da quanto abbiamo già detto, che la variabile `os.sep` viene impostata al tipo di barra separatrice che va bene per il sistema operativo sotto cui viene eseguito il programma.

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> calcFilePath.split(os.path.sep)  
['C:', 'Windows', 'System32', 'calc.exe']
```

Sotto sistemi OS X e Linux, all'inizio della lista restituita ci sarà una stringa vuota:

```
>>> '/usr/bin'.split(os.path.sep)  
[ "", 'usr', 'bin' ]
```

Il metodo stringa `split()` restituirà una lista contenente ciascuna parte del percorso. Funzionerà su qualsiasi sistema operativo, se gli passate `os.path.sep`.

## Trovare le dimensioni dei file e i contenuti delle cartelle

Una volta che avete dei modi per gestire i percorsi dei file, potete iniziare a raccogliere **informazioni su file e cartelle** specifici. Il modulo `os.path` offre funzioni per trovare le dimensioni di un file in byte e i file e le cartelle continuti in una data cartella.

- La chiamata `os.path.getsize(percuso)` restituirà le dimensioni in byte del file nell'argomento *percuso*.
- La chiamata `os.listdir(percuso)` restituirà una lista di stringhe con il nome di file di ciascun file nell'argomento *percuso*. (Notate che questa funzione è nel modulo `os`, non in `os.path`.)

Ecco quello che ottengo io quando provo queste funzioni nella shell interattiva:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')  
776192  
>>> os.listdir('C:\\Windows\\System32')  
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',  
--altre righe omesse--  
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

Come potete vedere, il programma *calc.exe* sul mio computer occupa 776.192 byte, e ho parecchi file nella cartella *C:\Windows\System32*. Se volessi sapere quanto occupano complessivamente tutti i file in questa directory, potrei usare `os.path.getsize()` e `os.listdir()` insieme.

```
>>> totalSize = 0
>>> for fileName in os.listdir('C:\\Windows\\System32'):
    totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32', fileName))

>>> print(totalSize)
1117846456
```

Ciclando su ciascun nome di file nella cartella *C:\Windows\System32*, la variabile `totalSize` viene incrementata con le dimensioni di ciascun file. Notate, quando chiamo `os.path.getsize()`, come uso `os.path.join()` per accodare al nome della cartella il nome del file corrente. L'intero che `os.path.getsize()` restituisce viene sommato al valore di `totalSize`.

Dopo aver ciclato su tutti i file, stampo `totalSize` per vedere le dimensioni complessive della cartella *C:\Windows\System32*.

## Verificare la validità di un percorso

Molte funzioni di Python andranno in crash con un errore, se fornite loro un **percorso che non esiste**. Il modulo `os.path` mette a disposizione funzioni per verificare se un dato percorso esiste e se indica un file o una cartella.

- La chiamata `os.path.exists(percuso)` restituisce True se il file o la cartella indicati nell'argomento esistono, restituisce False in caso contrario.
- La chiamata `os.path.isfile(percuso)` restituisce True se l'argomento *percuso* esiste ed è un file, restituisce False in caso contrario.
- La chiamata `os.path.isdir(percuso)` restituisce True se l'argomento *percuso* esiste ed è una cartella, restituisce False in caso contrario.

Ecco quello che ottengo io quando provo queste funzioni nella shell interattiva:

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

Potete stabilire se al momento al computer siano collegate una unità DVD o una chiavetta, utilizzando la funzione `os.path.exists()`. Per esempio, se volessi verificare se al mio computer Windows è collegata una chiavetta il cui volume è *D:\*, potrei fare così:

```
>>> os.path.exists('D:\\')
```

False

Oops! Sembra mi sia dimenticato di collegare la penna USB al computer.

## Il processo di lettura e scrittura di file

Non appena vi sentirete a vostro agio nel lavorare con cartelle e percorsi relativi, potrete specificare dove si trovano i file da leggere e scrivere. Le funzioni esaminate nelle prossime sezioni si applicano a file di puro testo. I file di puro testo contengono solo i caratteri fondamentali, senza indicazioni di tipo di carattere, dimensioni o colore. I file di testo con estensione .txt o i file script Python con estensione .py sono esempi di file di puro testo. Possono essere aperti con Blocco note di Windows o TextEdit di OS X. I vostri programmi possono leggere facilmente i contenuti di file di puro testo e possono trattarli come un normale valore stringa.

I file binari sono tutti gli altri tipi di file, come i documenti degli elaboratori di testo, i PDF, le immagini, i fogli di calcolo e i programmi eseguibili. Se aprite un file binario in Blocco note o TextEdit, vedrete un pasticcio senza senso, come nella Figura 8.5.



Figura 8.5 - Il programma calc.exe aperto in Blocco Note.

Dato che ogni tipo di file binario deve essere gestito in modo particolare, questo libro non parlerà di lettura e scrittura diretta di file binari grezzi. Per fortuna, molti moduli rendono più facile lavorare con i file binari: ne esplorerete uno, il modulo `shelve`, più avanti nel corso di questo capitolo. Per leggere e scrivere file in Python bisogna seguire tre passi.

1. Chiamare la funzione `open()` per avere di ritorno un oggetto `File`.
2. Chiamare il metodo `read()` o `write()` sull'oggetto `File`.
3. Chiudere il file chiamando il metodo `close()` sull'oggetto `File`.

## Apertura di file con la funzione `open()`

Per aprire un file con la funzione `open()`, passate alla funzione una stringa di percorso (assoluto o relativo) che indica il file che volette aprire. La funzione `open()` restituisce un oggetto `File`.

Provate a creare un file di testo con il nome `hello.txt` utilizzando Blocco note oppure TextEdit. Scrivete Hello world! come contenuto del file e salvatelo nella vostra cartella principale d'utente. Poi, se usate Windows, inserite nella shell interattiva:

```
>>> helloFile = open('C:\\\\Users\\\\vostra_cartella_home\\\\hello.txt')
```

Se usate OS X, inserite invece:

```
>>> helloFile = open('/Users/vostra_cartella_home/hello.txt')
```

Non dimenticate di sostituire a *vostra\_cartella\_home* il vostro nome utente per il computer. Per esempio, il mio nome utente è *asweigart*, perciò dovrei scrivere, sotto Windows, 'C:\\\\Users\\\\asweigart\\\\hello.txt'.

Entrambi questi comandi apriranno il file in modalità “lettura di puro testo”, o **modalità lettura** per brevità. Quando un file viene aperto in modalità lettura, Python vi consente solamente di leggere i dati da quel file; non potete scrivervi o modificarlo in alcun modo. La modalità lettura è quella predefinita per i file che si aprono in Python. Se non volete basarvi sui default di Python, potete specificare esplicitamente la modalità passando il valore stringa 'r' (da *read*, leggere) come secondo argomento di *open()*. Così, *open('/Users/asweigart/hello.txt', 'r')* e *open('/Users/asweigart/hello.txt')* sono espressioni equivalenti.

La chiamata di *open()* restituisce un oggetto **File**, che rappresenta un file sul vostro computer; è semplicemente un altro tipo di valore in Python, un po' come le liste e i dizionari che già conoscete. Nell'esempio precedente, l'oggetto **File** è memorizzato nella variabile *helloFile*. Ora, ogni volta che volete leggere o scrivere su quel file, potete farlo chiamando metodi sull'oggetto **File** in *helloFile*.

## Leggere i contenuti dei file

Ora che avete un oggetto **File**, potete iniziare a **leggere** da quel file. Se volete leggere **tutto il contenuto** di un file come un **valore stringa**, usate il metodo *read()* dell'oggetto **file**. Inserite quanto segue nella shell interattiva:

```
>>> helloContent = helloFile.read()  
>>> helloContent  
'Hello world!'
```

Se pensate i contenuti di un file come un unico grande valore stringa, il metodo *read()* restituisce la stringa che è memorizzata nel file.

In alternativa, potete usare il metodo *readlines()* per avere una lista di valori stringa dal file, una stringa per ogni riga di testo. Per esempio, create un file con il nome *sonnet29.txt* nella stessa directory di *hello.txt* e scrivete questo testo:

```
When, in disgrace with fortune and men's eyes,  
I all alone beweep my outcast state,  
And trouble deaf heaven with my bootless cries,  
And look upon myself and curse my fate,
```

Ricordatevi di separare le quattro righe di testo con a capo, poi inserite quanto segue nella shell interattiva:

```
>>> sonnetFile = open('sonnet29.txt')  
>>> sonnetFile.readlines()  
[When, in disgrace with fortune and men's eyes,\\n', ' I all alone beweep my  
outcast state,\\n', ' And trouble deaf heaven with my bootless cries,\\n', ' And  
look upon myself and curse my fate,']
```

Notate che ciascuno dei valori stringa termina con un carattere di a capo, \n, tranne l'ultima riga del file. Spesso è più facile lavorare con una lista di stringhe che con un unico, grande valore stringa.

## Scrivere su file

Python consente di **scrivere contenuti** in un file in modo simile a come la funzione `print()` “scrive” stringhe sullo schermo. Però non potete scrivere in un file che avete aperto in modalità lettura. Dovete aprirlo in modalità “scrittura testo puro” o “accodamento testo puro”, per brevità **modalità scrittura e modalità accodamento**.

La modalità scrittura sovrascriverà il file esistente e partirà da zero, come quando si sovrascrive il valore di una variabile con un nuovo valore. Passate 'w' (da *write*, scrivere) come secondo argomento a `open()` per aprire il file in modalità scrittura. La modalità accodamento, invece, accoderà il testo alla fine del file esistente. Potete pensarlo come l'accodamento di una lista in una variabile, al posto della sovrascrittura della variabile. Passate 'a' (da *append*, accodare) come secondo argomento a `open()` per aprire il file in modalità accodamento.

Se il nome di file passato a `open()` non esiste, le modalità scrittura e accodamento creeranno un **nuovo file vuoto**. Dopo aver letto o scritto un file, chiamate il metodo `close()` prima di aprire di nuovo il file. Uniamo tutti questi concetti. Inserite quanto segue nella shell interattiva:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

Come prima cosa, apriamo *bacon.txt* in modalità scrittura. Poiché non esiste ancora un file *bacon.txt*, Python lo crea. Chiamando `write()` sul file aperto e passando a `write()` l'argomento stringa 'Hello world! /n' la stringa viene scritta nel file e la funzione restituisce il numero dei caratteri scritti, compreso l'a capo. Poi il file viene chiuso.

Per **aggiungere testo** ai contenuti già esistenti del file, invece di sostituire la stringa appena scritta, apriamo il file in modalità accodamento. Scriviamo 'Bacon is not a vegetable.' sul file e lo chiudiamo. Infine, per stampare i contenuti del file su schermo, apriamo il file nella modalità lettura predefinita, si chiama `read()`, si memorizza il risultante oggetto File in `content`, si chiude il file e si stampa `content`.

Notate che il metodo `write()` non accoda automaticamente un carattere di a capo alla fine della stringa, come fa la funzione `print()`. Dovrete aggiungere voi questo carattere.

## Salvare variabili con il modulo shelve

Potete salvare variabili nei vostri programmi Python in file *shelf* (letteralmente “scaffale”) binari

utilizzando il modulo `shelve`. In questo modo, il vostro programma può ripristinare i dati nelle variabili dal disco fisso. Il modulo `shelve` vi consentirà di aggiungere funzioni *Salva* e *Apri* al vostro programma. Per esempio, se eseguite un programma e inserite qualche impostazione di configurazione, potete salvare queste impostazioni in un file shelf e poi fare in modo che il programma le carichi alla prossima esecuzione.

Inserite quanto segue nella shell interattiva:

```
>>> import shelve  
>>> shelfFile = shelve.open('mydata')  
>>> cats = ['Zophie', 'Pooka', 'Simon']  
>>> shelfFile['cats'] = cats  
>>> shelfFile.close()
```

Per leggere e scrivere dati utilizzando il modulo `shelve`, prima importate `shelve`. Chiamate `shelve.open()` e gli passate un nome di file, poi memorizzate il valore `shelf` in una variabile. Potete apportare modifiche al valore `shelf` come se fosse un valore in un dizionario. Quando avete finito, chiamate `close()` sul valore `shelf`. Qui, il nostro valore `shelf` è memorizzato in `shelfFile`.

Creiamo una lista `cats` e scriviamo `shelfFile['cats'] = cats` per memorizzare la lista in `shelfFile` come valore associato con la chiave '`cats`' (come in un dizionario). Poi chiamiamo `close()` su `shelfFile`.

Dopo aver eseguito il codice precedente in Windows, vedrete tre nuovi file nella directory di lavoro corrente: `mydata.bak`, `mydata.dat`, `mydata.dir`. In OS X, verrà creato solo un file `mydata.db`.

Questi file binari contengono i dati che avete memorizzato nel vostro `shelf`. Il formato di questi file binari non è importante; basta sapere quello che fa `shelve`, non come lo fa. Il modulo vi toglie ogni preoccupazione in merito a come memorizzare in un file i dati del vostro programma.

I vostri programmi possono usare il modulo `shelve` per poi riaprire e recuperare i dati da questi file `shelf`. I valori `shelf` non debbono essere aperti in modalità lettura o scrittura: possono operare in entrambi i modi una volta che sono stati aperti. Inserite quanto segue nella shell interattiva:

```
>>> shelfFile = shelve.open('mydata')  
>>> type(shelfFile)  
<class 'shelve.DbfilenameShelf'>  
>>> shelfFile['cats']  
['Zophie', 'Pooka', 'Simon']  
>>> shelfFile.close()
```

Qui, apriamo i file `shelf` per controllare che i nostri dati siano stati memorizzati correttamente. Inserendo `shelfFile['cats']` si ha di ritorno la stessa lista memorizzata in precedenza, così da verificare che sia stata salvata correttamente, poi si può chiamare `close()`.

Come i dizionari, i valori `shelf` hanno metodi `keys()` e `values()` che restituiscono in forma simile a una lista le chiavi e i valori nello `shelf`. Dato che questi metodi restituiscono valori simili a liste ma non vere liste, dovete passarli alla funzione `list()` per averli in forma di lista. Inserite quanto segue nella shell interattiva:

```
>>> shelfFile = shelve.open('mydata')  
>>> list(shelfFile.keys())  
['cats']  
>>> list(shelfFile.values())  
[['Zophie', 'Pooka', 'Simon']]
```

```
>>> shelfFile.close()
```

Il puro testo è utile per creare file leggibili in un editor come Blocco note oTextEdit, ma se volete salvare dati dai vostri programmi Python utilizzate il modulo `shelve`.

## Salvare variabili con la funzione `pprint.pformat()`

Come ricorderete dalla sezione “[Una stampa elegante](#)” a pagina 101, la funzione `pprint pprint()` offre il “pretty printing” sullo schermo dei contenuti di una lista o di un dizionario, mentre la funzione `pprint.pformat()` restituisce lo stesso testo sotto forma di stringa, invece di stamparlo. Non solo questa stringa è formattata in modo da renderne facile la lettura, ma è anche codice Python sintatticamente corretto. Supponiamo che abbiate un dizionario memorizzato in una variabile e che vogliate salvare questa variabile e i suoi contenuti, per poterli utilizzare in seguito. Usando `pprint.pformat()` avrete una stringa che potete scrivere in un file `.py`. Questo file sarà un vostro modulo personalizzato che potrete importare ogni volta che vorrete usare la variabile che vi è memorizzata.

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"['{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

Qui importiamo `pprint` per poter usare `pprint.pformat()`. Abbiamo una lista di dizionari, memorizzata in una variabile `cats`. Per avere disponibile la lista in `cats` anche dopo aver chiuso la shell, usiamo `pprint.pformat()` perché ce la restituisca come una stringa. Una volta avuti i dati in `cats` come una stringa, è facile scriverla in un file, che chiameremo `myCats.py`.

I moduli che un enunciato `import` importa sono a loro volta solo script Python. Quando la stringa ottenuta da `pprint.pformat()` è salvata in un file `.py`, quel file è un modulo che può essere importato come qualsiasi altro.

Poiché poi gli script Python non sono altro che file di testo con l'estensione di file `.py`, i vostri programmi Python possono addirittura generare altri programmi Python. Poi potete importare questi file negli script.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

Il vantaggio di creare un file `.py` (rispetto a salvare variabili con il modulo `shelve`) è che, trattandosi di un file di testo, i suoi contenuti possono essere letti e modificati da chiunque con un semplice editor di testo. Per la maggior parte delle applicazioni, però, il salvataggio dei dati con il modulo `shelve` è il modo d'elezione per salvare variabili in un file. Solo i tipi di dati fondamentali, come interi, virgola

mobile, stringhe, liste e dizionari, possono essere scritti in un file come testo puro; gli oggetti File, invece, per esempio, non possono essere codificati come testo.

## Progetto: generazione di file per test casuali

Supponiamo che siate un insegnante di geografia; avete in aula 35 studenti e volete assegnare loro un test relativo alle capitali degli Stati degli USA. Purtroppo c'è chi se ne approfitta, e non potete fidarvi che nessuno bari. Vorreste rendere casuale l'ordine delle domande, in modo che ogni test sia diverso dagli altri, rendendo così impossibile copiare semplicemente la successione delle risposte. Ovviamente, fare una cosa del genere manualmente sarebbe un'operazione lunga e noiosa, ma per fortuna sapete utilizzare un po' Python. Ecco che cosa fa il programma:

- crea 35 test differenti;
- crea 50 domande a scelta multipla per ciascun test, in ordine casuale;
- elenca la risposta corretta e tre risposte errate scelte a caso per ciascuna domanda, disposte in un ordine casuale;
- scrive i test in 35 file di testo;
- scrive le chiavi di risposta in 35 file di testo.

Questo significa che il codice dovrà fare quanto segue:

- memorizzare gli stati e le loro capitali in un dizionario;
- chiamare `open()`, `write()` e `close()` per i file di testo dei test e delle risposte;
- usare `random.shuffle()` per rendere casuale l'ordine delle domande e delle risposte a scelta multipla.

## Passo 1: memorizzare i dati del test in un dizionario

Il primo passo è creare lo scheletro dello script e popolarlo con i dati per il test. Create un file con il nome `randomQuizGenerator.py`, fatto in questo modo:

```

#! python3
# randomQuizGenerator.py - Crea test con domande e risposte in ordine casuale
# non ché le chiavi di risposta.

❶ import random

# I dati per i test. Le chiavi sono gli Stati, i valori le relative capitali.
❷ capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas':
'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine':
'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan':
'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri':
'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada':
'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New
Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh',
'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City',
'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence',
'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

# Genera 35 file con i test.
❸ for quizNum in range(35):
    # Da FARE: Creare i file con il test e le risposte.

    # Da FARE: Scrivere l'intestazione per il test.

    # Da FARE: Cambiare l'ordine degli stati.

    # Da FARE: Ciclare su tutti i 50 Stati, creando una domanda per ciascuno.

```

Dato che questo programma deve disporre in un ordine casuale domande e risposte, dovete importare il modulo `random` ❶ per poter utilizzare le sue funzioni. La variabile `capitals` ❷ contiene un dizionario con gli Stati degli USA come chiavi e le loro capitali come valori. Poiché volete creare 35 test, il codice che genera effettivamente i file con i test e le risposte (indicato qui dai commenti DA FARE, per il momento) andrà in un ciclo `for` che itererà per 35 volte ❸. (Questo numero ovviamente può essere modificato, per generare un numero qualsiasi di file con il test.)

## Passo 2: creare il file con il test e variare l'ordine delle domande

Ora dobbiamo mettere un po' di sostanza in quei DA FARE.

Il codice nel ciclo verrà ripetuto 35 volte, una per ciascun test, perciò all'interno del ciclo dovete preoccuparvi di un solo test alla volta. Innanzitutto creerete l'effettivo file del test: dovrà avere un nome di file univoco e dovrà anche contenere qualche tipo di intestazione standard, con spazi in cui lo studente potrà inserire nome, data e classe. Poi dovrete ottenere una lista di Stati in ordine casuale, che poi sarà utilizzata per creare le domande e risposte del test.

Aggiungete le righe di codice seguenti al file `randomQuizGenerator.py`:

```

#! python3
# randomQuizGenerator.py - Crea test con domande e risposte in ordine casuale
# nonché le chiavi di risposta.

-- righe omesse --

# Genera 35 file con i test.
for quizNum in range(35):
    # Crea i file con i test e le risposte.
❶    quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
❷    answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')

    # Scrive l'intestazione per il test.
❸    quizFile.write('Nome:\n\nData:\n\nClasse:\n\n')
    quizFile.write(( ' ' * 20) + 'Test sulle capitali USA (Modulo %s)' % (quizNum + 1))
    quizFile.write('\n\n')

    # Modifica l'ordine degli Stati.
states = list(capitals.keys())
❹    random.shuffle(states)

# Da FARE: Ciclare su tutti i 50 Stati, creando una domanda per ciascuno.

```

I nomi dei file per i test saranno *capitalsquiz<N>.txt*, dove <N> è un numero univoco che identifica il test, ricavato da quizNum, il contatore del ciclo for. Le risposte per *capitalquiz<N>.txt* saranno memorizzate in un file con nome *capitalsquiz\_Answers<N>.txt*. Ogni volta che viene attraversato il ciclo, il segnaposto %s in 'capitalsquiz%s.txt' e 'capitalsquiz\_answers%s.txt' verrà sostituito da (quizNum + 1), perciò il primo test e le relative risposte saranno in *capitalsquiz1.txt* e *capitalsquiz\_answers1.txt*. Questi file verranno creati con chiamate alla funzione `open()` in ❶ e ❷, con 'w' come secondo argomento, per aprirli in modalità scrittura.

Gli enunciati `write()` in ❸ creano una intestazione, con campi che lo studente dovrà compilare. Infine, viene creata una lista di Stati, in ordine casuale, con l'aiuto della funzione `random.shuffle()` ❹, che riordina in modo casuale i valori di qualsiasi lista che le venga passata come argomento.

### Passo 3: creare le risposte multiple

Ora dovete generare le risposte multiple per ciascuna domanda. Dovrete creare un altro ciclo for, questa volta per generare i contenuti per ciascuna delle 50 domande del test. Poi ci sarà un terzo ciclo for annidato, per generare le scelte multiple per ciascuna domanda. Ecco come dovete modificare il codice:

```

#! python3
# randomQuizGenerator.py - Crea test con domande e risposte in ordine casuale
# nonché le chiavi di risposta.

-- righe omesse --

# Cicla su tutti i 50 Stati, creando una domanda per ciascuno.
for questionNum in range(50):

    # Ottiene le risposte giuste e sbagliate.
    ❶ correctAnswer = capitals[states[questionNum]]
    ❷ wrongAnswers = list(capitals.values())
    ❸ del wrongAnswers[wrongAnswers.index(correctAnswer)]
    ❹ wrongAnswers = random.sample(wrongAnswers, 3)
    ❺ answerOptions = wrongAnswers + [correctAnswer]
    ❻ random.shuffle(answerOptions)

    # DA FARE: Scrivere le domande e le riposte multiple nel file del test.

    # DA FARE: Scrivere le risposte corrette in un file.

```

La risposta corretta è facile da ottenere: è già memorizzata come valore nel dizionario capitals **❶**. Questo ciclo itera sugli Stati nella lista riordinata states, da states[0] a states[49], trova ciascuno Stato in capitals e memorizza quindi la capitale di quello Stato in correctAnswer.

La lista delle possibili risposte sbagliate è un poco più complicata. Potete ottenerla duplicando tutti i valori nel dizionario capitals **❷**, eliminando la risposta corretta **❸** e selezionando a caso tre valori dalla lista **❹**. La funzione random.sample() semplifica questa operazione di selezione. Il suo primo argomento è la lista da cui volete selezionare; il secondo argomento è il numero dei valori che volete selezionare. La lista completa delle opzioni per le risposte è la combinazione di queste tre risposte errate con la risposta corretta **❺**.

Infine, le risposte devono essere ordinate in modo casuale **❻**, così che la risposta giusta non sia sempre la scelta D.

## Passo 4: scrivere i contenuti nei file del test e delle risposte

Tutto quello che resta è scrivere le domande sul file del test e le risposte nel file delle risposte. Ecco come dovete completare il vostro codice:

```

#! python3
# randomQuizGenerator.py - Crea test con domande e risposte in ordine casuale
# nonché le chiavi di risposta.

-- righe omesse --

# Cicla su tutti i 50 Stati, creando una domanda per ciascuno.
for questionNum in range(50):
    -- righe omesse --

    # Scrive domanda e risposte multiple nel file del test.
    quizFile.write('%s. Qual è la capitale dello Stato %s?\n' % (questionNum + 1,
        states[questionNum]))
    for i in range(4):
        quizFile.write(' %s. %s\n' % ('ABCD'[i], answerOptions[i]))
    quizFile.write('\n')

    # Scrive le risposte corrette in un file.
    answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[
        answerOptions.index(correctAnswer)]))
quizFile.close()
answerKeyFile.close()

```

u  
v  
w

Un ciclo for che itera sugli interi da 0 a 3 scriverà le varie risposte possibili nella lista answerOptions ①. L'espressione ABCD[i] in ② tratta la stringa 'ABCD' come un array e avrà come valore 'A', 'B', 'C' e infine 'D' nelle successive iterazioni del ciclo.

Nella riga finale ③, l'espressione answerOptions.index(correctAnswer) troverà l'indice intero della risposta corretta fra le risposte multiple ordinate in modo casuale, e 'ABCD'[answerOptions.index(correctAnswer)] avrà come valore la lettera corrispondente alla risposta corretta, che dovrà essere scritta nel file delle risposte.

Dopo aver eseguito il programma, ecco come si presenterà il file *capitalsquiz1.txt*, anche se ovviamente per voi domande e risposte multiple potranno essere diverse da quelle mostrate qui, a seconda dell'esito delle chiamate a random.shuffle():

Nome:

Data:

Classe:

### Test sulle capitali USA (Modulo 1)

1. Qual è la capitale dello Stato West Virginia?
  - A. Hartford
  - B. Santa Fe
  - C. Harrisburg
  - D. Charleston
2. Qual è la capitale dello Stato Colorado?
  - A. Raleigh
  - B. Harrisburg
  - C. Denver
  - D. Lincoln

-- e così via --

Il corrispondente file di testo *capitalsquiz\_answers1.txt* avrà questo aspetto:

1. D
  2. C
  3. A
  4. C
- e così via--

## Progetto: Appunti multipli

Supponiamo che abbiate il noioso compito di compilare molti moduli in una pagina web o in un software con molti campi di testo. Gli Appunti vi evitano di scrivere lo stesso testo ripetutamente, ma negli Appunti può trovarsi solo un frammento di testo alla volta; se avete più elementi testuali che dovete copiare e incollare, dovete continuare a evidenziare e copiare le stesse cose.

Potete scrivere un programma Python per **conservare vari elementi testuali**. Questi “Appunti multipli” o “multiclipboard” avranno il nome *mcb.pyw* (“mcb” è una abbreviazione di “multiclipboard”). L'estensione *.pyw* significa che Python non visualizzerà una finestra di Terminale quando esegue il programma. (Vedete l'[Appendice B](#) per ulteriori dettagli.)

Il programma salverà ogni frammento di testo sotto una parola chiave. Per esempio, se eseguite `py mcb.pyw save spam`, i contenuti correnti degli Appunti saranno salvati con la parola chiave `spam`. questo testo poi può essere caricato nuovamente negli Appunti eseguendo `py mcb.pyw spam`.

Se poi l'utente dimentica quali parole chiave ha salvato, può eseguire `py mcb.pyw list` per copiare negli Appunti un elenco di tutte le parole chiave.

Ecco quello che fa il programma:

- viene controllato l'argomento della riga di comando per trovare la parola chiave;
- se l'argomento è `save`, i contenuti degli Appunti vengono salvati in quella parola chiave;
- se l'argomento è `list`, vengono copiate negli Appunti tutte le parole chiave;
- altrimenti viene copiato alla tastiera il testo per la parola chiave.

Questo significa che il codice dovrà fare le cose seguenti:

- leggere gli argomenti della riga di comando da `sys.argv`;
- leggere e scrivere negli Appunti;
- salvare e caricare un file shelf.

Se usate Windows, potete eseguire facilmente questo script dalla finestra *Eseguire...* creando un file batch con il nome *mcb.bat* e con questi contenuti:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

## Passo 1: commenti e impostazione dello shelf

Iniziamo creando uno scheletro dello script con alcuni commenti e alcune impostazioni di base.

```

#! python3
# mcb.pyw - Salva e carica dagli Appunti frammenti di testo.
❶ # Uso: py.exe mcb.pyw save <parola chiave> - Salva gli Appunti nella parola chiave.
# py.exe mcb.pyw <parola chiave> - Carica negli Appunti la parola chiave.
# py.exe mcb.pyw list - Carica negli Appunti l'elenco delle parole chiave.

❷ import shelve, pyperclip, sys

❸ mcbShelf = shelve.open('mcb')

# DA FARE: Salvare i contenuti degli Appunti.

# DA FARE: Elencare le parole chiave e caricare i contenuti.

mcbShelf.close()

```

È pratica comune inserire informazioni generali d'uso nelle prime righe del file ❶. Se per caso vi dimenticate come eseguire il vostro script, potete sempre riguardare questi commenti. Poi importate i vostri moduli ❷.

Le operazioni di copia e incolla richiederanno il modulo `pyperclip` e la lettura degli argomenti da riga di comando richiederà il modulo `sys`. Farà comodo anche il modulo `shelve`: quando l'utente vorrà salvare un nuovo frammento di testo dagli Appunti, verrà salvato in un file shelf. Poi, quando l'utente vorrà incollare nuovamente il testo negli Appunti, si aprirà il file shelf e lo si caricherà nuovamente nel programma. Il file shelf avrà un nome con il prefisso *mcb* ❸.

## Passo 2: salvare i contenuti degli Appunti con una parola chiave

Il programma fa cose diverse a seconda che l'utente voglia salvare testo in una parola chiave, carica testo negli Appunti oppure vedere un elenco di tutte le parole chiave esistenti. Cominciamo con il primo caso.

Ecco come deve essere il vostro codice:

```

#! python3
# mcb.pyw - Salva e carica dagli Appunti frammenti di testo.
--righe omesse--

# Salva i contenuti degli Appunti.
❶ if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
    mcbShelf[sys.argv[2]] = pyperclip.paste()
❷ elif len(sys.argv) == 2:
    # DA FARE: Elencare le parole chiave e caricarne i contenuti.

mcbShelf.close()

```

Se il primo argomento da riga di comando (che si troverà sempre all'indice 1 della lista `sys.argv`) è 'save' ❶, il secondo argomento da riga di comando è la parola chiave per i contenuti correnti degli Appunti. La parola chiave sarà usata come chiave per `mcbShelf` e il valore relativo sarà il testo che si trova al momento negli Appunti ❷.

Se c'è un solo argomento da riga di comando, si da per scontato che sia o 'list' o una parola chiave di cui caricare i contenuti negli Appunti. Implementerete questo codice più avanti. Per ora, inserite solo

## Passo 3: elencare le parole chiave e caricare il contenuto di una parola chiave

Infine, implementiamo i due casi rimanenti. L'utente vuole caricare testo negli Appunti da una parola chiave, oppure vuole un elenco di tutte le parole chiave esistenti.

```
#! python3
# mcb.pyw - Salva e carica dagli Appunti frammenti di testo.
--righe omesse--

# Salva i contenuti degli Appunti.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
    mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
    # Elenca le parole chiave e ne carica i contenuti.
❶ if sys.argv[1].lower() == 'list':
   ❷     pyperclip.copy(str(list(mcbShelf.keys())))
   ❸ elif sys.argv[1] in mcbShelf:
        pyperclip.copy(mcbShelf[sys.argv[1]])

mcbShelf.close()
```

Se esiste un solo argomento da riga di comando, verifichiamo innanzitutto se sia 'list' ❶. Se è così, negli Appunti verrà copiata una rappresentazione stringa della lista delle chiavi di shelf. L'utente può incollare questa lista in un editor di testo per leggerla.

Altrimenti, si può dare per scontato che l'argomento da riga di comando sia una parola chiave. Se questa parola chiave esiste in mcbShelf come chiave, si può caricarne il valore negli Appunti ❸.

E questo è tutto. Il lancio di questo programma comporta passi diversi a seconda del sistema operativo del vostro computer. Consultate l'[Appendice B](#) per i dettagli.

Ricordate il programma creato nel [Capitolo 6](#), che conservava le password in un dizionario? Aggiornare le password richiedeva una modifica del codice sorgente del programma, il che non è l'ideale, perché l'utente medio non si sente molto a proprio agio nel modificare del codice sorgente per aggiornare il proprio software. Inoltre, ogni volta che si modifica il codice sorgente di un programma, si corre il rischio di introdurre accidentalmente nuovi errori. Memorizzando i dati per un programma in una posizione diversa dal codice, si può fare in modo che i programmi risultino più facili da usare per gli altri e più resistenti agli errori.

## Riepilogo

I **file** sono organizzati in **cartelle** (chiamate anche **directory**) e un **percorso** descrive la posizione di un file. Ogni programma che gira sul vostro computer ha una sua **directory di lavoro corrente**, che permette di specificare percorsi di file relativamente alla posizione corrente, invece di scrivere sempre il percorso completo (o assoluto). Il modulo `os.path` ha molte funzioni per manipolare i percorsi di file.

I vostri programmi possono anche interagire direttamente con i contenuti di file di testo. La funzione `open()` può **aprire** questi **file** per **leggerne il contenuto** come un'unica grande stringa, con il metodo `read()`, oppure come una lista di stringhe, con il metodo `readlines()`. La funzione `open()` può aprire i file in due **modalità** diverse, **scrittura** o **accodamento**, per creare nuovi file di testo o per aggiungere

contenuti a file di testo esistenti, rispettivamente.

Nei capitoli precedenti, avete usato gli Appunti per inserire grandi quantità di testo in un programma, anziché riscriverlo da zero. Ora potete fare in modo che i vostri programmi leggano file direttamente dal disco fisso, il che è un grande miglioramento, poiché i file sono molto meno volatili degli Appunti.

Nel prossimo capitolo vedrete come gestire i file stessi, copiandoli, cancellandoli, cambiando loro nome, spostandoli e facendo altre cose ancora.

## Domande di ripasso

1. A che cosa è relativo un percorso relativo?
2. Con che cosa inizia un percorso assoluto?
3. Che cosa fanno le funzioni `os.getcwd()` e `os.chdir()`?
4. Che cosa sono le cartelle `.` e `..`?
5. In `C:\bacon\eggs\spam.txt`, quale parte è il nome della directory e quale parte è il nome base?
6. Quali sono i tre argomenti “modalità” che possono essere passati alla funzione `open()`?
7. Che cosa succede, se un file esistente viene aperto in modalità scrittura?
8. Qual è la differenza fra i metodi `read()` e `readlines()`?
9. A quale struttura di dati assomiglia un valore `shelf`?

## Un po' di pratica

Per esercitarvi, progettate e scrivete i programmi seguenti.

## Estendere gli Appunti multipli

Estendete il programma per gli Appunti multipli visto in questo capitolo, in modo che abbia un argomento da riga di comando `delete <parola chiave>` che elimini una parola chiave dallo shelf. Poi aggiungete un argomento da riga di comando `delete` che elimini *tutte* le parole chiave.

## Mad Libs

Create un programma Mad Libs che legga file di testo e permetta all’utente di inserire testo a piacere ovunque nel file compaiano le parole *AGGETTIVO*, *NOME*, *AVVERBIO* o *VERBO*. Per esempio, il file di testo potrebbe essere fatto in questo modo:

Il panda AGGETTIVO se ne è andato verso NOME e poi VERBO. NOME che si trovava lì vicino non si è accorto di nulla.

Il programma troverà quelle occorrenze e inviterà l’utente a sostituirle:

Inserisci un aggettivo (di genere maschile, singolare):

**stupido**

Inserisci un nome con il suo articolo (determinato o indeterminato):

**il candelie re**

Inserisci un verbo (terza persona singolare, passato prossimo):

**ha urlato**

Inserisci un nome con il suo articolo (determinato o indeterminato):

**un camion dei pompieri**

Verrà creato allora questo file:

Il panda stupido se ne è andato verso il candeliere e poi ha urlato. Un camion dei pompieri che si trovava lì vicino non si è accorto di nulla.

I risultati devono essere stampati sullo schermo e salvati in un nuovo file di testo. [In inglese un programma simile è più semplice, perché non bisogna fare attenzione al genere e al numero: “silly”, per esempio, è un aggettivo che concorda con qualsiasi nome, maschile o femminile, singolare o plurale; “screamed” come verbo concorda con soggetti singolari e plurali e così via. Per l’italiano bisogna tener conto di molte possibilità, che devono essere catturate dal codice, oppure bisogna dare istruzioni abbastanza dettagliate, se si vuole che il testo finale sia sintatticamente corretto, anche se il significato può essere ridicolo.]

## Ricerca con espressioni regolari

Scrivete un programma che apra tutti i file .txt in una cartella e identifichi tutte le righe che corrispondono a un’espressione regolare fornita dall’utente. I risultati devono essere stampati sullo schermo.

# Organizzare i file

Nel capitolo precedente, avete visto come **creare e scrivere** nuovi **file** in Python. I vostri programmi possono anche **organizzare file** già esistenti sul disco fisso. Può darsi vi sia già capitato di dover scorrere una cartella con decine, centinaia o addirittura migliaia di file, per copiarli, cambiarne il nome, spostarli e comprimerli, tutto a mano.

Oppure, pensate ad attività come queste:

- fare copia di tutti i file PDF (e solo dei file PDF) in tutte le sottocartelle di una cartella;
- eliminare gli zeri all'inizio della numerazione dei nomi di file per tutti i file di una cartella contenente centinaia di file con nomi come *spam001.txt*, *spam002.txt*, *spam003.txt* e così via;
- comprimere i contenuti di varie cartelle in un unico file ZIP (che potrebbe essere un semplice sistema di backup).

Compiti così noiosi implorano proprio a gran voce di essere automatizzati in Python. Se programmate il vostro computer perché svolga queste attività, potete trasformarlo in un assistente veloce e che non commette mai errori.

Iniziando a lavorare con i file, vi sarà utile poter vedere rapidamente le estensioni dei nomi di file (*.txt*, *.pdf*, *.jpg* ecc.). In OS X e Linux con tutta probabilità le finestre del sistema mostrano automaticamente le estensioni, ma in Windows è possibile che le estensioni siano nascoste per impostazione predefinita. Per poterle vedere, dovete aprire **Start > Pannello di controllo > Opzioni cartella** e nella scheda *Visualizzazione* deselezionare l'opzione *Nascondi le estensioni per i tipi di file conosciuti*.

## Il modulo shutil

Il modulo `shutil` (che sta per “shell utilities”) mette a disposizione funzioni per **copiare**, **spostare**,

**rinominare e cancellare** file nei programmi Python. Per usare le funzioni di `shutil`, dovete prima importare il modulo con `import shutil`.

## Copiare file e cartelle

Il modulo `shutil` mette a disposizione funzioni per **copiare file** e anche intere **cartelle**.

La chiamata `shutil.copy(origine, destinazione)` copierà il file che si trova al percorso *origine* nella cartella al percorso *destinazione*, (Sia *origine* che *destinazione* sono stringhe). Se *destinazione* è un nome di file, verrà usata come nuovo nome del file copiato. Questa funzione restituisce una stringa con il percorso del file copiato.

Inserite quanto segue nella shell interattiva per vedere come funziona `shutil.copy()`.

```
>>> import shutil, os  
>>> os.chdir('C:\\')  
❶ >>> shutil.copy('C:\\spam.txt', 'C:\\delicious')  
'C:\\delicious\\spam.txt'  
❷ >>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')  
'C:\\delicious\\eggs2.txt'
```

La prima chiamata a `shutil.copy()` copia il file *C:\spam.txt* nella cartella *C:\delicious*. Il valore restituito è il percorso del file appena copiato. Notate che, essendo stato specificata come destinazione una cartella ❶, come nome di file del file copiato viene usato lo stesso nome originale, *spam.txt*. Anche la seconda chiamata a `shutil.copy()` ❷ copia il file *C:\eggs.txt* nella cartella *C:\delicious*, ma dà al file copiato il nome *eggs2.txt*.

Mentre `shutil.copy()` copia un singolo file, `shutil.copytree()` copia un'intera cartella e tutti i suoi contenuti. Chiamando `shutil.copytree(origine, destinazione)` si copierà la cartella con il percorso *origine*, con tutti i suoi file e le sue sottocartelle, nella cartella con percorso *destinazione*. I parametri *origine* e *destinazione* sono stringhe. La funzione restituisce una stringa con il percorso della cartella copiata. Inserite nella shell interattiva:

```
>>> import shutil, os  
>>> os.chdir('C:\\')  
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')  
'C:\\bacon_backup'
```

La chiamata a `shutil.copytree()` crea una nuova cartella con il nome *bacon\_backup* con gli stessi contenuti della cartella originale *bacon*.

## Spostare file e cartelle e cambiare il nome

Una chiamata a `shutil.move(origine, destinazione)` **sosterà il file o la cartella** con percorso *origine* al percorso *destinazione* e restituirà una stringa con il percorso assoluto della nuova posizione.

Se *destinazione* punta a una cartella, il file *origine* viene spostato nella *destinazione* e mantiene il nome che ha. Per esempio, inserite nella shell interattiva:

```
>>> import shutil  
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')  
'C:\\eggs\\bacon.txt'
```

Dando per scontato che nella directory *C:\* esista già una cartella con il nome *eggs*, questa chiamata a `shutil.move()` dice: “Sposta *C.\bacon.txt* nella cartella *C:\eggs*”.

Se nella cartella *C:\eggs* ci fosse già un file *bacon.txt*, questo verrebbe sovrascritto. Poiché è facile in questo modo sovrascrivere file senza volerlo, dovete sempre fare molta attenzione nell’usare `move()`.

Il percorso *destinazione* può specificare anche un nome di file. Nell’esempio seguente, il file *origine* viene spostato e gli viene cambiato il nome.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')  
'C:\\eggs\\new_bacon.txt'
```

Questa riga dice: “Sposta *C:\bacon.txt* nella cartella *C:\eggs* e, intanto che ci sei, cambia il nome del file da *bacon.txt* a *new\_bacon.txt*”.

Entrambi gli esempi precedenti funzionavano nell’ipotesi che già esistesse una cartella *eggs* nella directory *C:\*; se invece così non fosse, `move()` cambierebbe il nome del file *bacon.txt* in *eggs*.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')  
'C:\\eggs'
```

Qui, `move()` non riesce a trovare una cartella con il nome *eggs* nella directory *C:\* e perciò assume che *destinazione* specifichi un nome di file, non una cartella. Quindi il file di testo *bacon.txt* prende il nuovo nome di *eggs* (un file di testo senza l’estensione *.txt*) – il che probabilmente non è quel che volevate. Questo può essere un errore difficilmente individuabile in un programma, perché la chiamata `move()` può fare tranquillamente qualcosa di molto diverso da quello che vi aspettavate. Questo è un altro motivo per stare molto attenti, quando si usa `move()`.

Infine, le cartelle che costituiscono la destinazione devono già esistere, altrimenti Python presenterà un errore. Inserite quanto segue nella shell interattiva:

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')  
Traceback (most recent call last):  
  File "C:\\Python34\\lib\\shutil.py", line 521, in move  
    os.rename(src, real_dst)  
FileNotFoundError: [WinError 3] The system cannot find the path specified:  
'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):  
  File "<pyshell#29>", line 1, in <module>  
    shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')  
  File "C:\\Python34\\lib\\shutil.py", line 533, in move  
    copy2(src, real_dst)  
  File "C:\\Python34\\lib\\shutil.py", line 244, in copy2  
    copyfile(src, dst, follow_symlinks=follow_symlinks)  
  File "C:\\Python34\\lib\\shutil.py", line 108, in copyfile  
    with open(dst, 'wb') as fdst:  
FileNotFoundError: [Errno 2] No such file or directory: 'c:\\does_not_exist\\eggs\\ham'
```

Python cerca *eggs* e *ham* nella directory *does\_not\_exist*. Non trova questa directory, perciò non può spostare *spam.txt* al percorso che è stato specificato.

## Eliminare definitivamente file e cartelle

Potete **eliminare** un singolo file o una singola cartella vuota con funzioni del modulo `os`, mentre per eliminare una cartella e tutti i suoi contenuti bisogna usare il modulo `shutil`.

- Chiamando `os.unlink(percuso)` si cancellerà il file in `percuso`.
- Chiamando `os.rmdir(percuso)` si eliminerà la cartella in `percuso`. Questa cartella deve essere vuota, cioè non deve contenere né file né altre cartelle.
- Chiamando `shutil.rmtree(percuso)` si eliminerà la cartella in `percuso`, e verranno eliminati anche tutti i file e le cartelle che contiene.

Fate attenzione, se usate queste funzioni nei vostri programmi! Spesso è bene eseguire il programma trasformando queste chiamate in commenti e inserendo delle chiamate `print()` per vedere quali file verrebbero eliminati.

Ecco un esempio di un programma Python che è stato scritto per cancellare file con estensione `.txt`, ma che contiene un refuso (evidenziato in grassetto), per il quale vengono invece eliminati i file con estensione `.rxt`:

```
import os
for filename in os.listdir():
    if filename.endswith('.rxt'):
        os.unlink(filename)
```

Se ci fossero stati dei file importanti con estensione `.rxt`, involontariamente sarebbero stati eliminati definitivamente. Dovete prima eseguire il programma modificandolo in questo modo:

```
import os
for filename in os.listdir():
    if filename.endswith('.rxt'):
        #os.unlink(filename)
        print(filename)
```

Ora la chiamata `os.unlink()` è in un commento, perciò Python la ignora. Invece, stamperete il nome dei file che sarebbero stati eliminati. Facendo girare prima questa versione del programma potrete vedere che accidentalmente avete detto al programma di eliminare i file con estensione `.rxt` anziché quelli con estensione `.txt`.

Una volta sicuri che il programma funziona come volevate, potete cancellare la riga `print(filename)` ed eliminare il simbolo di commento dalla riga `os.unlink(filename)`. Poi potete eseguire nuovamente il programma per eliminare effettivamente i file.

## Eliminazione sicura con il modulo send2trash

Poiché la funzione interna di Python `shutil.rmtree()` elimina irreversibilmente file e cartelle, può essere pericoloso usarla. Un modo molto migliore per eliminare file e cartelle è utilizzare il modulo di terze parti `send2trash`.

Si può installare questo modulo eseguendo `pip install send2trash` da una finestra di Terminale.(Consultate l'[Appendice A](#) per una spiegazione approfondita di come installare moduli di terze parti.)

L'uso di `send2trash` è molto più sicuro delle normali funzioni di eliminazione di Python, perché **invia** cartelle e file **al Cestino**, invece di eliminarli definitivamente. Se un errore nel programma cancella

con send2trash qualcosa che non volevate eliminare, potete recuperarlo in seguito dal Cestino. Una volta installato send2trash, inserite quanto segue nella shell interattiva:

```
>>> import send2trash  
>>> baconFile = open('bacon.txt', 'a') # crea il file  
>>> baconFile.write('Bacon is not a vegetable.')  
25  
>>> baconFile.close()  
>>> send2trash.send2trash('bacon.txt')
```

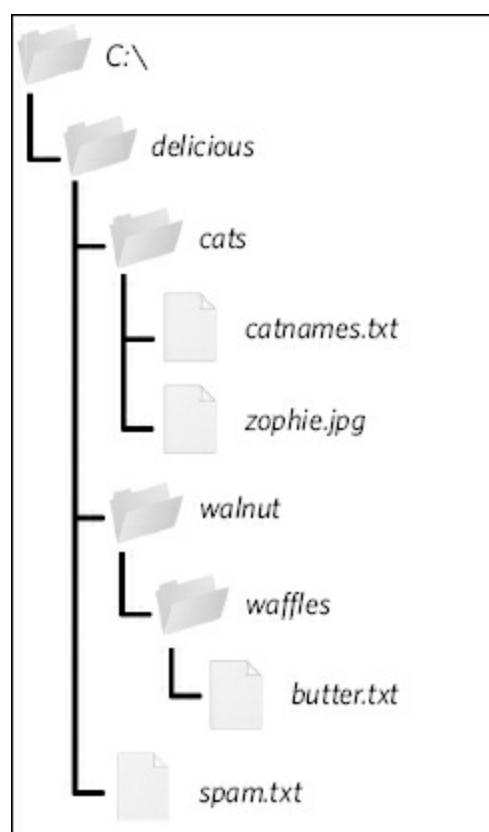
In generale, è bene usare sempre la funzione `send2trash.send2trash()` per eliminare file e cartelle. Inviare i file al Cestino permette di recuperarli in seguito, ma non libera spazio su disco come la loro eliminazione definitiva: se volete che il vostro programma liberi spazio su disco, utilizzate le funzioni di `os` e `shutil` per eliminare file e cartelle.

Notate che la funzione `send2trash()` può esclusivamente inviare file al Cestino, ma non può estrarli da lì.

## Percorrere un albero di directory

Supponiamo che vogliate cambiare il nome di tutti i file in qualche cartella e di tutti fili in ogni sottocartella di quella cartella. In altre parole, volete percorrere l'albero delle directory, modificando ogni file che incontrate. Scrivere un programma che faccia una cosa simile può essere complicato, ma per fortuna Python mette a disposizione una funzione che può gestire questo processo per voi.

Vediamo la cartella `C:\delicious` con i suoi contenuti rappresentata nella [Figura 9.1](#).



**Figura 9.1** - Un esempio di cartella che contiene tre sottocartelle e quattro file.

Ecco un esempio di programma che usa la funzione `os.walk()` sull'albero di directory della [Figura 9.1](#):

```

import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('La cartella corrente è ' + folderName)

    for subfolder in subfolders:
        print('SOTTOCARTELLA DI ' + folderName + ': ' + subfolder)

    for filename in filenames:
        print('FILE ALL'INTERNO DI ' + folderName + ': ' + filename)

print('')

```

Alla funzione `os.walk()` viene passato un singolo valore stringa, il percorso di una cartella. Si può usare `os.walk()` in un ciclo `for` per percorrere l'albero di directory, un po' come si può usare la funzione `range()` per percorrere un intervallo di numeri. A differenza di `range()`, però, la funzione `os.walk()` restituirà tre valori per ciascuna iterazione del ciclo:

1. una stringa con il nome della cartella corrente;
2. una lista di stringhe con le cartelle nella cartella corrente;
3. una lista di stringhe con i file nella cartella corrente.

(Per cartella corrente, si intende la cartella per l'iterazione in corso del ciclo `for`. La directory di lavoro corrente del programma non viene modificata da `os.walk()`).

Come si può scegliere il nome di variabile `i` nel codice `for i in range(10):`, si può scegliere anche i nomi di variabile per i tre valori di cui sopra. Io di solito uso i nomi `filename`, `subfolders` e `filenames`. Quando si eseguirà questo programma, l'output sarà di questo tipo:

```

La cartella corrente è C:\\delicious
SOTTOCARTELLA DI C:\\delicious: cats
SOTTOCARTELLA DI C:\\delicious: walnut
FILE ALL'INTERNO DI C:\\delicious: spam.txt

```

```

La cartella corrente è C:\\delicious\\cats
FILE ALL'INTERNO DI C:\\delicious\\cats: catnames.txt
FILE ALL'INTERNO DI C:\\delicious\\cats: zophie.jpg

```

```

La cartella corrente è C:\\delicious\\walnut
SOTTOCARTELLA DI C:\\delicious\\walnut: waffles

```

```

La cartella corrente è C:\\delicious\\walnut\\waffles
FILE ALL'INTERNO DI C:\\delicious\\walnut\\waffles: butter.txt.

```

Poiché `os.walk()` restituisce liste di stringhe per le variabili `subfolder` e `filename`, potete usare queste liste nei loro cicli `for`. Sostituite le chiamate alla funzione `print()` con il vostro codice personalizzato (o, se non avete bisogno di una o di entrambe, eliminate i relativi cicli `for`).

## Comprimere file con il modulo `zipfile`

Con tutta probabilità conoscete già i file ZIP (con estensione `.zip` del nome di file), che possono contenere versioni compresse di molti altri file.

La compressione dei file ne riduce le dimensioni, cosa utile quando li si deve trasmettere via

Internet. Dato poi che un file ZIP può contenere molti file e molte sottocartelle, costituisce un modo comodo per “impacchettare” molti file in un file unico. Questo file, chiamato anche **file archivio (archive)**, può poi essere allegato, per esempio, a un messaggio di posta elettronica.

I programmi Python possono sia creare sia aprire (ovvero estrarre) file ZIP utilizzando funzioni nel modulo `zipfile`. Supponiamo che abbiate un file ZIP con il nome `example.zip` con i contenuti rappresentati nella Figura 9.2. Potete scaricare questo file ZIP dall’indirizzo <http://nostarch.com/automatestuff/> o semplicemente seguire il testo utilizzando un file ZIP già presente sul vostro computer.

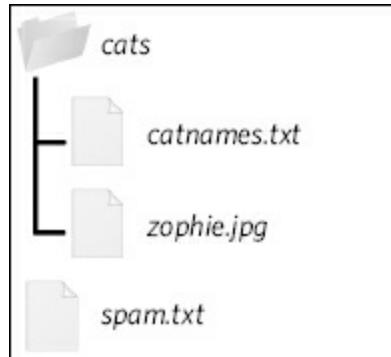


Figura 9.2 – I contenuti del file `example.zip`.

## Lettura di file ZIP

Per leggere i contenuti di un file ZIP, dovete prima creare un oggetto `ZipFile` (notate le due lettere, Z e F, maiuscole). Gli oggetti `ZipFile` sono concettualmente simili agli oggetti `File` che abbiamo già incontrato nel precedente capitolo, restituiti dalla funzione `open()`: sono i valori attraverso i quali il programma interagisce con il file. Per creare un oggetto `ZipFile`, chiamate la funzione `zipfile.ZipFile()`, passandole una stringa che contiene il nome del file `.zip`. Notate che `zipfile` è il nome del modulo Python mentre `ZipFile()` è il nome della funzione.

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> import zipfile, os
>>> os.chdir('C:\\\\') # passa alla cartella in cui si trova example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
❶ >>> 'Il file compresso è %s volte più piccolo!' % (round(spamInfo.file_size / spamInfo
.compress_size, 2))
'Il file compresso è 3.63 volte più piccolo!'
>>> exampleZip.close()
```

Un oggetto `ZipFile` ha un metodo `namelist()`, che restituisce una lista di stringhe per tutti i file e le cartelle contenuti nel file ZIP. Queste stringhe possono essere passate al metodo `getinfo()` per avere di ritorno un oggetto `ZipInfo` con informazioni su quel particolare file. Gli oggetti `ZipInfo` hanno i propri attributi, come `file_size` e `compress_size`, che danno le dimensioni in byte (numeri interi) del file originale e di quello compresso, rispettivamente. Mentre un oggetto `ZipFile` rappresenta un intero file archivio, un

oggetto ZipInfo contiene informazioni utili relative a un singolo file nell'archivio. L'istruzione in ❶ calcola l'efficienza della compressione, dividendo le dimensioni del file originale per quelle del file compresso, e stampa questa informazione utilizzando una stringa formattata con %s.

## Estrazione da file ZIP

Il metodo extractall() per gli oggetti ZipFile **estrae** tutti i **file** e le **cartelle** da un **file ZIP** nella directory di lavoro corrente.

```
>>> import zipfile, os  
>>> os.chdir('C:\\') # si sposta nella cartella che contiene example.zip  
>>> exampleZip = zipfile.ZipFile('example.zip')  
❶ >>> exampleZip.extractall()  
>>> exampleZip.close()
```

Dopo aver eseguito questo codice, i contenuti di *example.zip* saranno stati estratti in *C:\*. Opzionalmente, si può passare a extractall() il nome di una cartella, perché i file estratti vengano salvati in una cartella diversa dalla directory di lavoro corrente. Se la cartella passata al metodo extractall() non esiste, viene creata. Per esempio, se si sostituisce la chiamata in ❶ con exampleZip.extractall('C:\\delicious'), il codice estrae i file da *example.zip* e li salva in una cartella *C:\delicious* di nuova creazione.

Il metodo extract() per gli oggetti ZipFile **estrae** un singolo file dal file ZIP. Continuate l'esempio nella shell interattiva:

```
>>> exampleZip.extract('spam.txt')  
'C:\\spam.txt'  
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')  
'C:\\some\\new\\folders\\spam.txt'  
>>> exampleZip.close()
```

La stringa che passate a extract() deve corrispondere a una delle stringhe nella lista restituita da namelist(). Facoltativamente, si può passare a extract() un secondo argomento per estrarre il file salvandolo in una cartella diversa dalla directory di lavoro corrente. Se questo secondo argomento è una cartella che non esiste, Python la crea. Il valore restituito da extract() è il percorso assoluto in cui viene estratto il file.

## Creare file ZIP e aggiungervi file

Per **creare** i vostri **file ZIP** compressi, dovete **aprire** l'oggetto ZipFile in **modalità scrittura**, passando 'w' come secondo argomento. (Il procedimento è analogo all'apertura di un file di testo in modalità scrittura, con il passaggio di 'w' alla funzione open()). Se si passa un percorso al metodo write() di un oggetto ZipFile, Python comprime il file in quel percorso e lo aggiunge al file ZIP. Il primo argomento del metodo write() è una stringa contenente il nome del file da aggiungere. Il secondo argomento è il parametro del *tipo di compressione*, che dice al computer quale algoritmo usare per comprimere i file; potete sempre impostare questo valore semplicemente a zipfile.ZIP\_DEFLATED. (Questo specifica l'algoritmo di compressione *deflate*, che funziona bene con tutti i tipi di dati.) Inserite quanto segue nella shell interattiva:

```
>>> import zipfile  
>>> newZip = zipfile.ZipFile('new.zip', 'w')  
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)  
>>> newZip.close()
```

Questo codice creerà un nuovo file ZIP con il nome *new.zip*, con i contenuti compressi di *spam.txt*. Ricordate che, come quando si scrive su file, la modalità scrittura eliminerà tutti i contenuti esistenti nel file ZIP. Se volete semplicemente **aggiungere file** a un file ZIP già esistente, passate a `zipfile.ZipFile()` come secondo argomento '*a*', per aprire il file ZIP in **modalità accodamento**.

## Progetto: date da formato americano a europeo

Immaginate che il vostro capo vi invii migliaia di file con date in formato americano (mm-gg-aaaa) nel nome e che dobbiate trasformare le date in formato europeo (gg-mm-aaaa). A mano ci vorrebbe l'intera giornata! Proviamo a scrivere invece un programma che svolga automaticamente questo compito noioso.

Ecco che cosa fa il programma:

- cerca nella directory di lavoro corrente tutti i nomi di file che contengono date in formato americano;
- quando ne trova uno, cambia il nome al file scambiando mese e giorno in modo da portare la data nel formato europeo.

Questo significa che il codice dovrà fare queste cose:

- creare un'espressione regolare che possa identificare lo schema di testo delle date in formato americano;
- chiamare `os.listdir()` per trovare tutti i file nella directory di lavoro;
- ciclare su ciascun nome di file, utilizzando l'espressione regolare per verificare se contiene una data;
- se ha una data, cambiare il nome del file con `shutil.move()`.

Per questo progetto, aprite una nuova finestra di file editor e salvate il vostro codice con il nome *renameDate.py*.

## Passo 1: creare un'espressione per le date in formato americano

La prima parte del programma deve importare i moduli necessari e creare un'espressione regolare che possa identificare le date in formato mm-gg-aaaa. I commenti DA FARE vi ricorderanno che cosa rimane da realizzare in questo programma: scrivendoli con questa etichetta sarà più facile trovarli con il comando Ctrl-F di IDLE. Iniziate il vostro codice in questo modo:

```
#! python3
# renameDates.py - Prende i file il cui nome contiene una data in formato americano
# MM-GG-AAAA e lo trasforma nel formato GG-MM-AAAA.

❶ import shutil, os, re

❷ # Crea una espressione regolare che identifica i file con il formato di data americano.
datePattern = re.compile(r"""^(.*?)(0|1)?\d)-((0|1|2|3)?\d)-((19|20)\d\d)(.*?)$""", re.VERBOSE)

❸ # DA FARE: Cicla sui file nella directory di lavoro.
```

```
# DA FARE: Salta i file che non contengono una data.
```

```
# DA FARE: Prende le diverse parti del nome di file.
```

```
# DA FARE: Crea il nome di file in formato europeo.
```

```
# DA FARE: Prende il percorso completo, assoluto del file.
```

```
# DA FARE: Cambia il nome dei file.
```

Da quel che abbiamo visto in questo capitolo, sapete che si può usare la funzione `shutil.move()` per cambiare il nome di file: i suoi argomenti sono il nome del file a cui cambiare il nome e il nuovo nome. Questa funzione fa parte del modulo `shutil`, perciò dovete importare questo modulo ❶. Prima di cambiare nome ai file, dovete identificare i file su cui volete intervenire. Nomi di file con date, come *spam4-4-1984.txt* e *01-03-2014eggs.zip* devono cambiare nome, mentre file senza data, come *littlebrother.epub*, possono essere ignorati.

Potete usare un'espressione regolare per identificare questo schema. Dopo aver importato il modulo `re`, chiamate `re.compile()` per creare un oggetto Regex ❷. Passando come secondo argomento `re.VERBOSE` ❸ potrete inserire spazi e commenti nella stringa dell'espressione regolare, per renderla più leggibile.

La stringa dell'espressione regolare inizia con `^(.*?)` per identificare qualsiasi testo all'inizio del nome del file che preceda la data. Il gruppo `(0|1)?\d` corrisponde al mese. La prima cifra può essere 0 o 1, in modo che l'espressione individui 12 per dicembre ma anche 02 per febbraio. Questa cifra peraltro è facoltativa, in modo che il mese possa essere 04 oppure 4 per aprile. Il gruppo per il giorno è `((0|1|2|3)?\d)` e segue una logica simile: 3, 03 e 31 sono tutti numeri validi per i giorni. (Sì, questa espressione regolare accetterà anche date non valide come 4-31-2017, 2-29-2013 e 0-15-2014. Nelle date si incontrano molti casi speciali spinosi che è facile trascurare. Questa espressione regolare è semplice, tuttavia in questo programma funziona abbastanza bene.)

1885 è un anno valido, ma potete cercare solo anni del XX o XXI secolo. In questo modo il vostro programma non correrà il rischio di individuare accidentalmente anche nomi di file che non sono date pur avendo un formato simile a una data, come per esempio *10-10-1000.txt*.

La parte `(.*?)$` dell'espressione regolare indica qualsiasi testo dopo la data.

## Passo 2: identificare le parti della data nei nomi di file

Poi il programma deve ciclare sulla lista delle stringhe dei nomi di file restituita da `os.listdir()` e confrontarli con l'espressione regolare. Qualsiasi nome che non contenga una data deve essere trascurato. Per i nomi di file che contengono una data, il testo corrispondente verrà memorizzato in più variabili. Realizzate le prime tre parti DA FARE nel vostro programma, con il codice seguente:

```
#! python3
# renameDates.py - Prende i file il cui nome contiene una data in formato americano
# MM-GG-AAAA e lo trasforma nel formato GG-MM-AAAA.

--righe omesse--

# Cicla sui file nella directory di lavoro.
for amerFilename in os.listdir('.'):
    mo = datePattern.search(amerFilename)

    # Salta i file senza data.
❶    if mo == None:
        continue

❷    # Prende le diverse parti del nome di file.
    beforePart = mo.group(1)
    monthPart = mo.group(2)
    dayPart = mo.group(4)
    yearPart = mo.group(6)
    afterPart = mo.group(8)

--righe omesse--
```

Se l'oggetto Match restituito dal metodo `search()` è `None` ❶, allora il nome del file in `amerFilename` non corrisponde all'espressione regolare. L'enunciato `continue` ❷ salterà il resto del ciclo e passerà al nome di file successivo.

Altrimenti, le varie stringhe che corrispondono ai gruppi dell'espressione regolare vengono memorizzate in variabili con i nomi `beforePart`, `monthPart`, `dayPart`, `yearPart` e `afterPart` ❸. Le stringhe in queste variabili saranno poi usate, nel passo successivo, per formare il nome di file in formato europeo.

Per non perdervi fra i numeri dei gruppi, provate a leggere l'espressione regolare dall'inizio e contate ogni volta che incontrate una parentesi di apertura. Senza pensare al codice, scrivete solo l'ossatura dell'espressione regolare. Questo può aiutarvi a visualizzare i gruppi. Per esempio:

```
datePattern = re.compile(r"""\^(1)      # tutto il testo prima della data
                           (2 (3) )-          # una o due cifre per il mese
                           (4 (5) )-          # una o due cifre per il giorno
                           (6 (7) )           # quattro cifre per l'anno
                           (8)$              # tutto il testo dopo la data
""", re.VERBOSE)
```

Qui, i numeri da 1 a 8 rappresentano i gruppi nell'espressione regolare che avete scritto. Scrivendo l'ossatura dell'espressione regolare, con le parentesi e i numeri dei gruppi solamente, potete avere un'idea migliore della vostra espressione regolare, prima di proseguire con il resto del programma.

## Passo 3: formare il nuovo nome e assegnarlo al file

Come passo finale, concatenate le stringhe nelle variabili del passo precedente con la data in formato europeo: il giorno viene prima del mese. Realizzate le ultime tre parti DA FARE del vostro programma con questo codice:

```
#! python3
# renameDates.py - Prende i file il cui nome contiene una data in formato americano
# MM-GG-AAAA e lo trasforma nel formato GG-MM-AAAA.

--righe omesse--

❶ # Forma il nome di file in formato europeo.
❷ euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart + afterPart

❸ # Prende il percorso completo, assoluto del file.
absWorkingDir = os.path.abspath('.')
amerFilename = os.path.join(absWorkingDir, amerFilename)
euroFilename = os.path.join(absWorkingDir, euroFilename)

❹ # Cambia il nome dei file.
❺ print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))
❻ #shutil.move(amerFilename, euroFilename)      # togliere il commento dopo il test
```

Memorizzate la stringa concatenata in una variabile euroFilename ❶. Poi, passate il nome di file originale, che si trova in amerFilename e la nuova variabile euroFilename alla funzione shutil.move(), per cambiare il nome del file ❷.

Questo programma ha la chiamata shutil.move() in un commento e al suo posto stampa i nomi dei file a cui verrebbe cambiato il nome ❸. Eseguire prima il programma in questa forma consente di verificare che i file vengano rinominati correttamente. Poi potete togliere il commento alla chiamata shutil.move() ed eseguire il programma per cambiare effettivamente il nome dei file.

## Idee per programmi simili

Esistono molti altri motivi per cui potreste voler cambiare il nome di un gran numero di file:

- per aggiungere un prefisso all'inizio del nome del file, per esempio aggiungere *spam\_* e cambiare il nome di *eggs.txt* in *spam\_eggs.txt*;
- per cambiare nomi di file con date in formato europeo in nomi con date in formato americano;
- per eliminare gli zeri da nomi di file come *spam0042.txt*.

## Progetto: backup di una cartella in un file ZIP

Supponiamo che stiate lavorando a un progetto e che teniate i file relativi in una cartella che si chiama *C:\AlsPythonBook*.

Avete paura di poter perdere il vostro lavoro, perciò vorreste creare delle "istantanee" dell'intera cartella in file ZIP. Vorreste conservarne varie versioni, perciò vorreste che il nome del file ZIP comprenda un numero che si incrementa ogni volta, per esempio *AlsPythonBook\_1.zip*, *AlsPythonBook\_2.zip*, *AlsPythonBook\_3.zip* e così via. Potreste farlo a mano, ma è abbastanza seccante e potreste anche accidentalmente sbagliare a numerare i file ZIP. Sarebbe molto più semplice eseguire un programma che svolga per voi questa noiosa attività.

Per questo progetto, aprirete una nuova finestra di file editor e salvate il file con il nome

## Passo 1: stabilire il nome del file ZIP

Il codice per questo programma sarà inserito in una funzione `backupToZip()`. In questo modo sarà facile copiare e incollare la funzione in altri programmi Python in cui serva questa funzionalità. Alla fine del programma, la funzione verrà chiamata per eseguire il backup.

Iniziate il vostro programma in questo modo:

```
#! python3
# backupToZip.py - Copia una cartella con tutti i suoi contenuti in un file ZIP
# il cui nome di file viene ogni volta incrementato.

❶ import zipfile, os

def backupToZip(folder):
    # Effettua il backup dei contenuti di "folder" in un file ZIP.

    folder = os.path.abspath(folder) # make sure folder is absolute

    # Stabilisce, in base ai nomi dei file già esistenti,
    # quale debba essere il nome del nuovo file.
    ❷ number = 1
    ❸ while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

    ❹ # DA FARE: Crea il file ZIP.

    # DA FARE: Percorre tutto l'albero della cartella e comprime tutti i file.
    print('Fatto.')

    backupToZip('C:\\\\delicious')
```

Innanzitutto le cose di base: la riga che descrive che cosa fa il programma, poi l'enunciato per importare i moduli `zipfile` e `os` ❶.

Definite una funzione `backupToZip` che prende un solo parametro, `folder`. Questo parametro è una stringa con il percorso alla cartella dei cui contenuti volete effettuare il backup. La funzione determinerà quale nome usare per il file ZIP che creerà; poi creerà il file, percorrerà la cartella `folder` e aggiungerà al file ZIP tutte le sottocartelle e i file. Scrivete commenti DA FARE per questi passi nel codice sorgente, per ricordarvi di realizzarli in seguito ❹.

La prima parte, dare un nome al file ZIP, usa il nome base del percorso assoluto di `folder`. Se la cartella di cui si deve fare il backup è `C:\\delicious`, il nome del file ZIP deve essere `delicious_N.zip`, dove  $N = 1$  la prima volta che si esegue il programma,  $N = 2$  la seconda volta, e via di questo passo. Potete stabilire che cosa debba essere  $N$  verificando se esiste già `delicious_1.zip`, poi se esiste già `delicious_2.zip` e così via. Usate una variabile `number` per  $N$  ❷, e continuate a incrementarla nel ciclo che chiama `os.path.exists()` per verificare se il file esiste già ❸. Il primo nome di file di cui si stabilisce che non esiste farà sì che il ciclo arrivi a `break`: avete trovato il nome del nuovo file zip.

## Passo 2: creare il nuovo file ZIP

Ora creiamo il file ZIP. Modificate il vostro programma in questo modo:

```
#! python3
# backupToZip.py - Copia una cartella con tutti i suoi contenuti in un file ZIP
# il cui nome di file viene ogni volta incrementato.

--righe omesse--
while True:
    zipfilename = os.path.basename(folder) + '_' + str(number) + '.zip'
    if not os.path.exists(zipfilename):
        break
    number = number + 1

# Crea il file ZIP.
print('Crea %s...' % (zipfilename))
❶ backupZip = zipfile.ZipFile(zipfilename, 'w')

# DA FARE: Percorre tutto l'albero della cartella e comprime tutti i file.
print('Fatto.')

backupToZip('C:\\delicious')
```

Ora che il nome del nuovo file ZIP è memorizzato nella variabile `zipfilename`, potete chiamare `zipfile.ZipFile()` per creare effettivamente il file ZIP. Non dimenticate di passare '`w`' come secondo argomento, perché il file ZIP venga aperto in modalità scrittura.

## Passo 3: percorrere l'albero di directory e aggiungere elementi al file ZIP

Ora dovete usare la funzione `os.walk()` per elencare tutti i file nella cartella e nelle sue sottocartelle. Modificate così il vostro programma:

```
#! python3
# backupToZip.py - Copia una cartella con tutti i suoi contenuti in un file ZIP
# il cui nome di file viene ogni volta incrementato.

--righe omesse--
# Percorre tutto l'albero della cartella e comprime i file.
❶ for foldername, subfolders, filenames in os.walk(folder):
    print('Aggiungo i file in %s...' % (foldername))
    # Aggiunge la cartella corrente al file ZIP.
   ❷     backupZip.write(foldername)
    # Aggiunge tutti i file in questa cartella al file ZIP.
   ❸     for filename in filenames:
        newBase = os.path.basename(folder) + '_'
        if filename.startswith(newBase) and filename.endswith('.zip'):
            continue                # non aggiunge i file ZIP dei backup precedenti
        backupZip.write(os.path.join(foldername, filename))

❹ backupZip.close()
print('Fatto.')

backupToZip('C:\\delicious')
```

Potete usare `os.walk()` in un ciclo `for` ❶ e a ogni iterazione restituirà il nome di cartella corrente dell'iterazione, le sottocartelle in quella cartella e i nomi di file nella cartella.

Nel ciclo `for`, la cartella viene aggiunta al file ZIP ❷. Il ciclo `for` annidato può percorrere ciascun nome di file nella lista `filenames` ❸. Ciascuno viene aggiunto al file ZIP, tranne gli ZIP di backup effettuati in precedenza.

Quando eseguite il programma darà un output analogo a questo:

```
Creo delicious_1.zip...
Aggiungo i file in C:\delicious...
Aggiungo i file in C:\delicious\cats...
Aggiungo i file in C:\delicious\waffles...
Aggiungo i file in C:\delicious\walnut...
Aggiungo i file in C:\delicious\walnut\waffles...
Fatto.
```

La seconda volta che lo eseguite, metterà tutti i file di `C:\delicious` in un file ZIP con il nome `delicious_2.zip` e via di seguito.

## Idee per programmi simili

Potete percorrere un albero di directory e aggiungere file ad archivi ZIP compressi in molti altri programmi. Per esempio, potete scrivere programmi che facciano cose di questo genere:

- esplorino un albero di directory e archivino solo i file con determinate estensioni, per esempio `.txt` o `.py` e niente altro;
- esplorino un albero di directory e archivino tutti i file tranne quelli con estensione `.txt` e `.py`;
- trovino in un albero di directory la cartella che contiene il maggior numero di file o la cartella che occupa più spazio su disco.

## Riepilogo

Anche se siete utenti esperti, probabilmente gestite i file manualmente con mouse e tastiera. I sistemi operativi moderni rendono facile lavorare con pochi file, ma a volte bisogna svolgere qualche attività che con l'interfaccia grafica del sistema operativo richiederebbe ore.

I moduli `os` e `shutil` offrono **funzioni** per **copiare**, **spostare**, **rinominare** ed **eliminare file**. Quando eliminate file, è preferibile utilizzare il modulo `send2trash` per spostare i file nel cestino, anziché eliminarli definitivamente. Quando si scrivono programmi che gestiscono file, è una buona idea mettere in un **commento** il codice che esegue le effettive operazioni di copia/spostamento/cambiamento di nome/eliminazione e inserire invece una chiamata a `print()`, in modo da poter eseguire il programma e verificare esattamente su quali file vada a intervenire, prima magari di commettere qualche errore irreparabile.

Spesso avrete bisogno di svolgere operazioni di questo genere non solo su file in una cartella ma anche su ogni **cartella** in quella cartella, ogni cartella in quelle **sottocartelle** e così via. La funzione `os.walk()` gestisce questi percorsi fra le cartelle, in modo che possiate concentrarvi su quello che il programma deve fare con i file trovati.

Il modulo `zipfile` mette a disposizione un modo per **comprimere** ed **estrarre file** in archivi `.zip` direttamente attraverso Python. Combinato con le funzioni di gestione file in `os` e `shutil`, `zipfile` rende facile comprimere in un unico archivio molti file provenienti da qualunque parte del vostro disco

fisso. Questi file *.zip* sono molto più facili da caricare sui siti web o da inviare come allegati di posta, rispetto a molti file separati.

I capitoli precedenti di questo libro vi hanno presentato codice sorgente da copiare, ma quando scriverete i vostri programmi probabilmente non avrete un codice perfetto al primo colpo. Per questo il prossimo capitolo si concentra su alcuni moduli Python che vi aiuteranno ad analizzare il vostri programmi e a farne il debug, in modo da ottenere rapidamente una versione perfettamente funzionante.

## Domande di ripasso

1. Qual è la differenza fra `shutil.copy()` e `shutil.copytree()`?
2. Quale funzione si usa per cambiare nome ai file?
3. Qual è la differenza fra le funzioni di eliminazione nei moduli `send2trash` e `shutil`?
4. Gli oggetti `ZipFile` hanno un metodo `close()` analogo al metodo `close()` degli oggetti `File`. Quale metodo di `ZipFile` è equivalente al metodo `open()` degli oggetti `File`?

## Un po' di pratica

Per esercitarvi, scrivete programmi che svolgano le attività seguenti.

### Copia selettiva

Scrivete un programma che percorra un albero di cartelle e cerchi file con una determinata estensione (per esempio *.pdf* o *.jpg*). Copiate questi file, dovunque si trovino, in una nuova cartella.

### Eliminare file non necessari

Non è raro che pochi file non necessari ma di grandi dimensioni rubino molto spazio sul disco fisso. Se avete bisogno di liberare spazio sul disco, otterrete il massimo risultato con il minimo sforzo eliminando, fra i file non necessari, quelli più corposi. Prima però dovete identificarli.

Scrivete un programma che percorra un albero di cartelle e cerchi file o cartelle particolarmente grandi, poniamo di dimensioni superiori ai 100 MB. (Ricordate che per trovare le dimensioni di un file potete usare `os.path.getsize()` dal modulo `os`.) Stampate su schermo l'elenco di questi file con il loro percorso assoluto.

### Riempire i vuoti

Scrivete un programma che trovi tutti i file con un determinato prefisso, per esempio *spam001.txt*, *spam002.txt* e così via, in una stessa cartella e identifichi gli eventuali salti nella numerazione (per esempio, nel caso ci siano *spam001.txt* e *spam003.txt* ma non *spam002.txt*). Il programma deve cambiare il nome dei file successivi in modo da colmare le lacune.

Come ulteriore sfida, scrivete un altro programma che possa inserire lacune in una successione di file numerati, in modo da poter aggiungere un nuovo file.

# Debug

Ora che siete in grado di scrivere programmi più complessi, comincerete anche a incappare in **errori (“bug”)** non tanto facili da **identificare**. Questo capitolo esamina **strumenti e tecniche** per trovare la causa dei bug nei vostri programmi, per poterli eliminare più rapidamente e con minore fatica.

Per parafrasare una vecchia barzelletta da programmatori, “Scrivere il codice rappresenta il 90 per cento della programmazione. Il debug del codice spiega il rimanente 90 per cento”.

Il vostro computer farà solo quello che gli dite di fare; non vi leggerà nel pensiero e non farà quello che *volevate* facesse. Anche chi è programmatore di professione commette regolarmente errori, perciò non scoraggiatevi se il vostro programma ha un problema.

Per fortuna, esistono strumenti e tecniche per identificare quello che fa esattamente il vostro codice e stabilire che cosa non va. In primo luogo guarderete logging e asserzioni, due caratteristiche che possono aiutare a identificare presto gli errori: in generale, quanto prima si catturano gli errori, tanto più facile sarà risolverli.

In secondo luogo, vedrete come usare il debugger, una funzione di IDLE che esegue un programma una istruzione alla volta, dando la possibilità di esaminare i valori nelle variabili mentre il codice è in esecuzione e di seguire come cambiano i valori nel corso del programma. È molto più lento rispetto all'esecuzione normale del programma, ma è utile per vedere i valori effettivi mentre il programma gira, anziché dedurre quali siano quei valori guardando soltanto il codice sorgente.

## Eccezioni

Python “solleva un'eccezione” (in inglese si dice: “raises an exception”) ogni volta che si cerca di eseguire codice non valido. nel [Capitolo 3](#), avete visto come gestire le eccezioni di Python con enunciati try ed except, in modo che il vostro programma possa affrontare eccezioni che avete previsto. Potete però anche sollevare le vostre eccezioni nel vostro codice. Sollevare **un'eccezione** è un modo per dire “Smetti di eseguire il codice in questa funzione e trasferisci l'esecuzione del programma

all'enunciato except”.

Le eccezioni vengono sollevate con un enunciato raise. Nel codice, un enunciato raise è costituito da:

- la parola chiave raise;
- una chiamata alla funzione Exception();
- una stringa con un messaggio d'errore utile passata alla funzione Exception().

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> raise Exception('Questo è il messaggio di errore.')
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    raise Exception('Questo è il messaggio di errore.')
Exception: Questo è il messaggio di errore.
```

Se non vi sono enunciati try ed except che coprano l'enunciato raise che ha sollevato l'eccezione, il programma si blocca e visualizza il messaggio d'errore dell'eccezione.

Spesso non è la funzione stessa, ma il codice che chiama la funzione che sa come gestire un'eccezione. Così in genere vedrete un enunciato raise all'interno di una funzione e gli enunciati try ed except nel codice che chiama la funzione. Per esempio, apriete una nuova finestra di file editor, inserite il codice seguente e salvate il programma con il nome di *boxPrint.py*:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
        ❶      raise Exception('Il simbolo deve essere una stringa di un solo carattere.')
    if width <= 2:
        ❷      raise Exception('La larghezza deve essere maggiore di 2.')
    if height <= 2:
        ❸      raise Exception('L\'altezza deve essere maggiore di 2.')
    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)

    for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
        try:
            ❹            boxPrint(sym, w, h)
        except Exception as err:
            ❺            print('Si è verificata una eccezione: ' + str(err))
```

Qui abbiamo definito una funzione boxPrint(), che prende un carattere, una larghezza e un'altezza, e usa il carattere per creare una piccola immagine di un rettangolo di quella larghezza e altezza. Questa forma rettangolare viene stampata sullo schermo.

Vogliamo che il carattere sia un carattere singolo e che altezza e larghezza siano maggiori di 2. Aggiungiamo degli enunciati if per sollevare delle eccezioni se questi requisiti non sono soddisfatti. Poi, quando chiamiamo boxPrint() con vari argomenti, i nostri try/except gestiranno gli argomenti non validi.

Questo programma usa la forma except Exception as err dell'enunciato except ❻. Se da boxPrint() viene restituito un oggetto Exception ❺ ❻ ❾, questo enunciato except lo conserva in una variabile err. L'oggetto Exception poi può essere convertito in una stringa passandolo a str(), in modo da generare un messaggio di errore amichevole ❺. Se eseguite questo programma *boxPrint.py*, l'output sarà analogo

a questo:

```
****  
* *  
* *  
****  
00000000000000000000  
0 0  
0 0  
0 0  
00000000000000000000  
Si è verificata una eccezione: La larghezza deve essere maggiore di 2.  
Si è verificata una eccezione: Il simbolo deve essere una stringa di un solo carattere.
```

Utilizzando gli enunciati `try` ed `except`, potete gestire gli errori in modo più elegante, invece di lasciare semplicemente che il programma vada in crash.

## Ottenere il traceback come stringa

Quando Python incontra un errore, produce una grande quantità di informazioni sull'errore, che vanno sotto il nome di **traceback**.

Il traceback comprende il messaggio d'errore, il numero della riga che ha provocato l'errore e la sequenza delle chiamate di funzione che ha portato all'errore. Questa sequenza è determinata **stack di chiamata**.

Aprite una nuova finestra di file editor in IDLE, inserite il programma seguente e salvatelo con il nome *errorExample.py*:

```
def spam():  
    bacon()  
def bacon():  
    raise Exception('Questo è il messaggio di errore.')  
  
spam()
```

Quando si esegue *errorExample.py*, l'output sarà come questo:

```
Traceback (most recent call last):  
  File "errorExample.py", line 7, in <module>  
    spam()  
  File "errorExample.py", line 2, in spam  
    bacon()  
  File "errorExample.py", line 5, in bacon  
    raise Exception('Questo è il messaggio di errore.')  
Exception: Questo è il messaggio di errore.
```

Dal traceback, si può vedere che l'errore è avvenuto alla riga 5, nella funzione `bacon()`. Questa particolare chiamata a `bacon()` è arrivata dalla riga 2, nella funzione `spam()`, che a sua volta è stata chiamata nella riga 7. Nei programmi in cui le funzioni possono essere chiamate da molti punti diversi, lo stack di chiamata può aiutare a stabilire quale chiamata abbia portato all'errore.

Il traceback è visualizzato da Python ogni volta che un'eccezione sollevata non viene gestita. Potete ottenerlo anche come una stringa, chiamando `traceback.format_exc()`. Questa funzione è utile se volete le

informazioni del traceback di un'eccezione, ma volete anche che un enunciato `except` gestisca elegantemente l'eccezione. Dovrete importare il modulo `traceback` di Python prima di chiamare questa funzione.

Per esempio, invece di lasciare che il programma vada in crash non appena si verifica un'eccezione, potete scrivere le informazioni di traceback in un file di log e lasciare che il programma continui la sua esecuzione.

Potete esaminare il file di log in seguito, quando siete pronti al debug del programma. Inserite quanto segue nella shell interattiva:

```
>>> import traceback  
>>> try:  
    raise Exception('Questo è il messaggio di errore.')  
  
except:  
    errorFile = open('errorInfo.txt', 'w')  
    errorFile.write(traceback.format_exc())  
    errorFile.close()  
    print('Le informazioni di traceback sono state scritte nel file errorInfo.txt.')  
  
122
```

Le informazioni di traceback sono state scritte nel file `errorInfo.txt`.

Il 122 è il valore restituito dal metodo `write()`, poiché 122 sono i caratteri che sono stati scritti nel file. Il testo del traceback è stato scritto nel file `errorInfo.txt` in questo modo:

Traceback (most recent call last):

  File "<pyshell#12>", line 2, in <module>

Exception: Questo è il messaggio di errore.

## Asserzioni

Una **asserzione** è un controllo di sicurezza per verificare che il codice non stia facendo qualcosa di ovviamente sbagliato. Questi controlli di sicurezza sono eseguiti da enunciati `assert`. Se il controllo fallisce, viene sollevata un'eccezione `AssertionError`. Nel codice, un enunciato `assert` è formato da:

- la parola chiave `assert`;
- una condizione (cioè un'espressione che può avere valore `True` o `False`);
- una virgola;
- una stringa che deve essere visualizzata quando la condizione è `False`.

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> podBayDoorStatus = 'aperto'  
>>> assert podBayDoorStatus == 'aperto', 'Le porte devono essere in posizione "aperto".'  
>>> podBayDoorStatus = 'Mi dispiace non posso farlo.'  
>>> assert podBayDoorStatus == 'aperto', 'Le porte devono essere in posizione "aperto".'  
Traceback (most recent call last):  
  File "<pyshell#10>", line 1, in <module>  
    assert podBayDoorStatus == 'aperto', 'Le porte devono essere in posizione "aperto".'  
AssertionError: Le porte devono essere in posizione "aperto".
```

Abbiamo impostato `podBayDoorStatus` a 'aperto', e da quel momento ci aspettiamo che il valore della variabile sia 'aperto'. In un programma che usa questa variabile, potremmo aver scritto una gran quantità di codice dando per scontato che il valore sia 'aperto', cioè codice il cui buon funzionamento dipende dal fatto che sia 'aperto'. Perciò aggiungiamo un'asserzione per essere sicuri di non sbagliare nel presumere che `podBayDoorStatus` sia 'aperto'. Qui includiamo il messaggio 'Le porte devono essere in posizione "aperto"', in modo che sia facile vedere che cosa non va se l'asserzione fallisce.

In seguito, potremmo commettere l'errore di assegnare a `podBayDoorStatus` un altro valore, senza renderce conto in mezzo alle molte righe di codice. L'asserzione coglie l'errore e ci dice chiaramente che cosa c'è di sbagliato.

In parole povere, l'enunciato `assert` dice "Io asserisco che questa condizione è vera, e, se non lo è, c'è un errore da qualche parte nel programma". A differenza di quel che avviene per le eccezioni, il codice non deve gestire gli enunciati `assert` con `try` ed `except`; se un `assert` fallisce, il programma deve andare in crash. Fallendo rapidamente, si riduce il tempo che intercorre fra il momento in cui si verifica l'errore e il momento in cui ve ne accorgete: così si ridurrà anche la quantità di codice che dovrete controllare per poter trovare il punto che provoca l'errore.

Le asserzioni sono per gli errori del programmatore, non per quelli dell'utente. Per errori recuperabili (per esempio: un file non si trova, l'utente inserisce dati non validi), sollevate un'eccezione, anziché rilevarli con un enunciato `assert`.

## Uso di una asserzione in una simulazione di semaforo

Supponiamo che stiate preparando un programma che simula un semaforo. La struttura di dati che rappresenta i semafori a un incrocio è un dizionario con le chiavi 'ns' e 'ew', che stanno per i semafori rivolti in direzione nord-sud e est-ovest, rispettivamente. I valori di queste chiavi saranno una delle stringhe 'verde', 'giallo' o 'rosso'. Il codice sarà di questo tipo:

```
market_2nd = {'ns': 'verde', 'ew': 'rosso'}
mission_16th = {'ns': 'rosso', 'ew': 'verde'}
```

Queste due variabili saranno per l'incrocio fra Market Street e 2nd Street e fra Mission Street e 16th Street. Per iniziare il progetto, volette scrivere una funzione `switchLights()`, che prenda il dizionario di un incrocio come argomento e cambi il colore dei semafori.

Di primo acchito, potreste pensare che `switchLights()` debba semplicemente far passare ciascun colore al successivo nella sequenza: i valori 'verde' devono diventare 'giallo', i valori 'giallo' devono diventare 'rosso' e i valori 'rosso' devono diventare 'verde'. Il codice per implementare questa idea potrebbe essere come questo:

```
def switchLights(stoplight):
    for key in stoplight.keys():
        if stoplight[key] == 'verde':
            stoplight[key] = 'giallo'
        elif stoplight[key] == 'giallo':
            stoplight[key] = 'rosso'
        elif stoplight[key] == 'rosso':
            stoplight[key] = 'verde'

switchLights(market_2nd)
```

Avrete già visto sicuramente il problema, ma facciamo finta che invece abbiate scritto il resto del codice della simulazione, migliaia di righe, senza accorgervene. Quando finalmente eseguite la simulazione, il programma non va in crash, ma le vostre automobili virtuali non faranno che scontrarsi.

Poiché avete già scritto il resto del programma, non avete idea di dove possa essere l'errore: magari è nel codice che simula le auto o in quello che simula i guidatori virtuali. Potrebbero volerci ore per identificare l'errore nella funzione `switchLights()`. Se però, scrivendo quella funzione, aveste aggiunto un'asserzione per verificare che in ogni momento almeno uno dei semafori sia rosso, avreste potuto includere qualcosa di questo genere in fondo alla funzione:

```
assert 'red' in stoplight.values(), 'Nessun semaforo è rosso! ' + str(stoplight)
```

Con questa asserzione inserita, il programma andrebbe in crash con questo messaggio d'errore:

```
Traceback (most recent call last):
  File "carSim.py", line 14, in <module>
    switchLights(market_2nd)
  File "carSim.py", line 13, in switchLights
    assert 'rosso' in stoplight.values(), 'Nessun semaforo è rosso! ' + str(stoplight)
❶ AssertionError: Nessun semaforo è rosso! {'ns': 'giallo', 'ew': 'verde'}
```

La riga importante qui è `AssertionError` ❶. Che il vostro programma vada in crash non è l'ideale, ma questo fa capire immediatamente che un controllo di sicurezza è fallito: nessuna delle direzioni all'incrocio ha il semaforo rosso, quindi tutti potrebbero passare simultaneamente. Provocando presto un crash nel vostro programma, potete risparmiarvi una gran quantità di fatica nella ricerca degli errori.

## Disattivare le asserzioni

Le **asserzioni possono essere disattivate** passando l'opzione `-O` quando si manda in esecuzione Python. Questo va bene quando avete finito di scrivere e collaudare il vostro programma e non volete che sia rallentato dai controlli di sicurezza (anche se in genere gli enunciati `assert` non provocano variazioni apprezzabili di velocità). Le asserzioni sono per lo sviluppo, non per il prodotto finale. Quando consegnate il vostro programma a qualcun altro che dovrà utilizzarlo, deve essere privo di errori e non deve aver bisogno dei controlli di sicurezza. Consultate l'[Appendice B](#) per i particolari su come lanciare i vostri programmi probabilmente funzionanti con l'opzione `-O`.

## Logging

Se avete mai inserito un enunciato `print()` nel vostro codice per avere in output il valore di qualche variabile mentre il programma è in esecuzione, avete usato una forma di **logging** per mettere a punto il vostro codice. Il logging è un ottimo modo per capire che cosa succede nel vostro programma e in che ordine succede. Il modulo `logging` di Python facilita la creazione di una **registrazione dei messaggi personalizzati** che scrivete. Questi messaggi di log descriveranno quando l'esecuzione del programma ha raggiunto la chiamata della funzione di logging ed elencheranno le variabili che avete specificato in quel punto. L'assenza di un messaggio di log vi dirà che una parte del codice è stata saltata e non è mai stata eseguita.

## Uso del modulo logging

Per abilitare il modulo `logging` e visualizzare messaggi di log sullo schermo mentre il programma è in esecuzione, copiate quanto segue all'inizio del programma:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
```

Non dovete preoccuparvi troppo di come questo funziona, ma, fondamentalmente, quando Python registra un evento, crea un oggetto `LogRecord` che contiene informazioni su quell'evento. La funzione `basicConfig()` del modulo `logging` vi permette di specificare quali dettagli dell'oggetto `LogRecord` volete vedere e come volete che quei dettagli vengano visualizzati.

Supponiamo che abbiate scritto una funzione per calcolare il **fattoriale** di un numero. In matematica il fattoriale di 4 è  $1 \times 2 \times 3 \times 4$ , ovvero 24. Il fattoriale di 7 è  $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$ , ovvero 5040. Aprite una nuova finestra di file editor e inserite il codice che segue. Contiene un errore, ma inserirete anche vari messaggi di log che vi aiuteranno a capire che cosa non va. Salvate il programma con il nome *factorialLog.py*.

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
logging.debug('Inizio del programma')

def fattoriale(n):
    logging.debug('Inizio di fattoriale(%s%%)' % (n))
    total = 1
    for i in range(n + 1):
        total *= i
        logging.debug('i è ' + str(i) + ', il totale è ' + str(total))
    logging.debug('Fine di fattoriale(%s%%)' % (n))
    return total

print(fattoriale(5))
logging.debug('Fine del programma')
```

Qui, usiamo la funzione `logging.debug()` quando vogliamo stampare le informazioni di log. Questa funzione `debug()` chiama `basicConfig()` e stampa una riga di informazioni. Queste informazioni saranno nel formato che abbiamo specificato in `basicConfig()` e includeranno i messaggi che abbiamo passato a `debug()`. La chiamata `print(fattoriale(5))` fa parte del programma originale, perciò il risultato viene visualizzato anche se i messaggi di logging sono disattivati.

L'output del programma sarà di questo tipo:

```
2017-05-23 16:20:12,664 - DEBUG - Inizio del programma
2017-05-23 16:20:12,664 - DEBUG - Inizio di fattoriale(5)
2017-05-23 16:20:12,665 - DEBUG - i è 0, il totale è 0
2017-05-23 16:20:12,668 - DEBUG - i è 1, il totale è 0
2017-05-23 16:20:12,670 - DEBUG - i è 2, il totale è 0
2017-05-23 16:20:12,673 - DEBUG - i è 3, il totale è 0
2017-05-23 16:20:12,675 - DEBUG - i è 4, il totale è 0
2017-05-23 16:20:12,678 - DEBUG - i è 5, il totale è 0
2017-05-23 16:20:12,680 - DEBUG - Fine di fattoriale(5)
0
2017-05-23 16:20:12,684 - DEBUG - Fine del programma
```

La funzione `fattoriale()` restituisce 0 come fattoriale di 5, il che è ovviamente sbagliato. Il ciclo `for` dovrebbe moltiplicare il valore in total per i numeri da 1 a 5, ma i messaggi di log visualizzati da `logging.debug()` mostrano che la variabile `i` parte da 0 e non da 1. Poiché zero moltiplicato per qualsiasi numero è sempre zero, il resto delle iterazioni darà sempre il valore sbagliato. I messaggi di log forniscono una traccia che consente di stabilire quando le cose hanno cominciato ad andare storte. Modificate la riga `for i in range(n + 1):` in `for i in range(1, n + 1):` e poi eseguite nuovamente il programma. L'output ora sarà come questo:

```
2017-05-23 17:13:40,650 - DEBUG - Inizio del programma
2017-05-23 17:13:40,651 - DEBUG - Inizio di fattoriale(5)
2017-05-23 17:13:40,651 - DEBUG - i è 1, il totale è 1
2017-05-23 17:13:40,654 - DEBUG - i è 2, il totale è 2
2017-05-23 17:13:40,656 - DEBUG - i è 3, il totale è 6
2017-05-23 17:13:40,659 - DEBUG - i è 4, il totale è 24
2017-05-23 17:13:40,661 - DEBUG - i è 5, il totale è 120
2017-05-23 17:13:40,661 - DEBUG - Fine di fattoriale(5)
120
2017-05-23 17:13:40,666 - DEBUG - Fine del programma
```

La chiamata a `fattoriale(5)` ora restituisce correttamente 120. I messaggi di log hanno fatto capire che cosa succedeva all'interno del ciclo e questo ci ha portati direttamente a identificare l'errore. Potete vedere che le chiamate a `logging.debug()` hanno stampato non solo le stringhe che sono state passate loro, ma anche un orario e la parola `DEBUG`.

## Non fate il debug con `print()`

Scrivere `import logging` e poi `logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')` è un po' noioso. Potreste essere tentati di usare invece delle chiamate a `print()`, ma non cedete alla tentazione, altrimenti quando avrete messo a punto il programma vi ritroverete a perdere un sacco di tempo per eliminare le chiamate a `print()` per ogni messaggio di log. Potreste perfino eliminare accidentalmente qualche chiamata a `print()` che non era utilizzata per i messaggi di log. La cosa bella dei messaggi di log è che siete liberi di infarcire il vostro programma con tutti quelli che volete, poi potrete disattivarli aggiungendo una singola chiamata a `logging.disable(logging.CRITICAL)`. A differenza di `print()`, il modulo `logging` permette di passare facilmente dalla visualizzazione alla rimozione dei messaggi di log.

I messaggi di log servono al programmatore, non all'utente. All'utente non interessano i contenuti di qualche valore di dizionario che voi dovete vedere per orientarvi nella messa a punto. Nel caso di messaggi che l'utente vorrà vedere, come *File non trovato* o *Input non valido, per favore inserisci un numero*, dovete usare una chiamata a `print()`. Non dovete privare l'utente di informazioni che gli sono utili, dopo aver disattivato i messaggi di log.

## Livelli di logging

I *livelli di logging* offrono un modo per classificare per importanza i messaggi di log. Esistono cinque livelli di logging, descritti nella [Tabella 10.1](#), in ordine dal meno al più importante. I messaggi possono essere registrati a ciascun livello utilizzando una diversa funzione di logging.

Livello	Funzione di logging	Descrizione
DEBUG	logging.debug()	Il livello più basso. Si usa per piccoli dettagli. Di solito questi messaggi interessano solo quando si diagnosticano problemi.
INFO	logging.info()	Si usa per informazioni su eventi generali nel programma o per confermare che le cose funzionano in quel punto del programma.
WARNING	logging.warning()	Si usa per indicare un potenziale problema che non impedisce al programma di funzionare ma potrebbe farlo in futuro.
ERROR	logging.error()	Si usa per registrare un errore, a causa del quale il programma non ha potuto fare qualcosa.
CRITICAL	logging.critical()	Livello massimo. Si usa per indicare un errore fatale che ha impedito o sta per impedire totalmente l'esecuzione del programma.

Il vostro messaggio di logging viene passato a queste funzioni come una stringa. I livelli di logging sono solo suggerimenti: alla fine, sta a voi decidere in quale categoria un vostro messaggio rientri. Inserite quello che segue nella shell interattiva:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
>>> logging.debug('Qualche dettaglio di debug.')
2015-05-18 19:04:26,901 - DEBUG - Qualche dettaglio di debug.
>>> logging.info('Il modulo logging funziona.')
2015-05-18 19:04:35,569 - INFO - Il modulo logging funziona.
>>> logging.warning('Sta per essere registrato un messaggio di errore.')
2015-05-18 19:04:56,843 - WARNING - Sta per essere registrato un messaggio di errore.
>>> logging.error('Si è verificato un errore.')
2015-05-18 19:05:07,737 - ERROR - Si è verificato un errore.
>>> logging.critical('Il programma non può continuare!')
2015-05-18 19:05:45,794 - CRITICAL - Il programma non può continuare!
```

Il vantaggio dei livelli di logging è che è possibile modificare la priorità dei messaggi di logging che si vogliono vedere. Passando logging-DEBUG alla funzione basicConfig() come argomento per parola chiave level si vedranno i messaggi di tutti i livelli di logging (poiché DEBUG è il livello più basso). Dopo aver sviluppato ancora un po' il programma, però, forse sarete interessati solo agli errori. In quel caso, potete impostare l'argomento level di basicConfig() a logging.ERROR, così vedrete solo i messaggi ERROR e CRITICAL e non più quelli dei livelli DEBUG, INFO e WARNING.

## Disattivare il logging

Dopo che avete messo a punto il vostro programma, probabilmente non vorrete più che tutti questi messaggi di log vi ingombrino lo schermo. La funzione logging.disable() disattiva i messaggi in modo che non dobbiate entrare nel programma ed eliminare manualmente tutte le chiamate di logging. Semplicemente passate a logging.disable() un livello di logging, e questo non farà più apparire tutti i messaggi di log di quel livello o dei livelli inferiori. Se volete disattivare completamente i messaggi di logging, basta che aggiungiate al programma logging.disable(logging.CRITICAL). Per esempio, inserite

quanto segue nella shell interattiva:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s - %(levelname)s - %(message)s')
>>> logging.critical('Errore critico! Errore critico!')
2017-05-22 11:10:48,054 - CRITICAL - Errore critico! Errore critico!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Errore critico! Errore critico!')
>>> logging.error('Errore! Errore!')
```

Poiché `logging.disable()` disattiva tutti i messaggi che vengono dopo, probabilmente lo vorrete inserire vicino alla riga `import logging` nel codice del vostro programma. In questo modo, potete facilmente trovarlo per trasformarlo in un commento o riattivarlo per abilitare o disabilitare i messaggi di logging, a seconda delle necessità.

## Logging in un file

Anziché visualizzare i messaggi di log sullo schermo, potete scriverli in un **file di testo**. La funzione `logging.basicConfig()` prende un argomento per parola chiave `filename`:

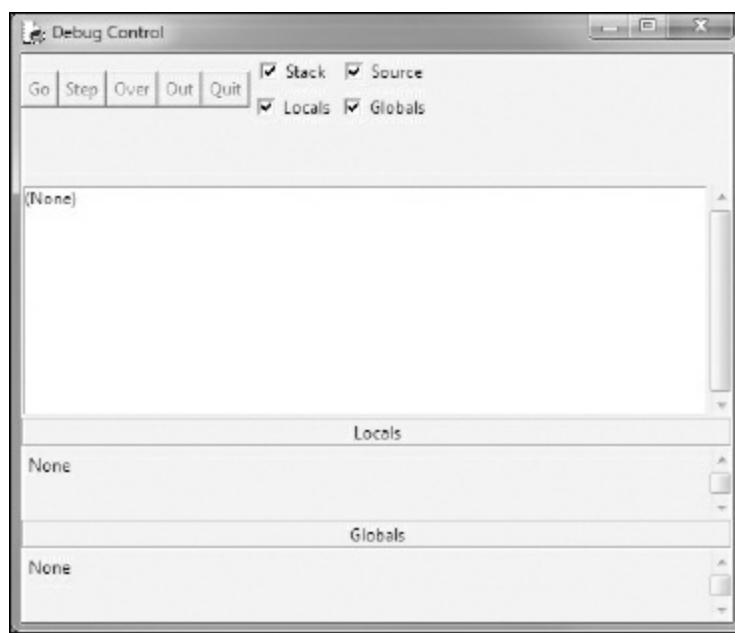
```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='
%(asctime)s - %(levelname)s - %(message)s')
```

I messaggi di log saranno salvati in *myProgramLog.txt*. Questi messaggi certo sono utili, ma ingombrano lo schermo e rendono difficile la lettura dell'output del programma. Scrivendoli in un file lo schermo resta pulito e i messaggi si possono leggere dopo l'esecuzione del programma. Il file può essere aperto in qualsiasi editor di testo.

## Debugger

Il debugger è una caratteristica di IDLE che permette di eseguire un programma una riga alla volta. Il debugger esegue una riga di codice, poi aspetta che gli dicate di continuare. Eseguendo il programma “sotto debugger” in questo modo, potete prendervi tutto il tempo che volete per esaminare i valori delle variabili in qualunque punto. Si tratta di uno strumento prezioso per individuare gli errori.

Per attivare il debugger di IDLE, selezionate **Debug > Debugger** nella finestra della shell interattiva. Questo comando aprirà la finestra *Debug Control* ([Figura 10.1](#)).



**Figura 10.1** - La finestra Debug Control.

Quando compare la finestra *Debug Control*, selezionate tutte le quattro caselle di controllo *Stack*, *Locals*, *Source* e *Globals*, così che la finestra presenti tutte le informazioni di debug. Mentre questa finestra è aperta, ogni volta che eseguite un programma dal file editor, il debugger fermerà l'esecuzione prima della prima istruzione e visualizzerà:

- la riga di codice che sta per essere eseguita;
- un elenco di tutte le variabili locali e dei loro valori;
- un elenco di tutte le variabili globali e dei loro valori.

Noterete che nell'elenco delle variabili globali compaiono molte variabili che non avete definito, per esempio `__builtins__`, `__doc__`, `__file__`, e così via. Sono variabili che Python imposta automaticamente ogni volta che esegue un programma. Il significato di queste variabili va al di là degli obiettivi di questo libro, e potete tranquillamente ignorarle.

Il programma rimarrà in pausa finché non premete uno dei cinque pulsanti presenti nella finestra *Debug Control*: *Go*, *Step*, *Over*, *Out* o *Quit*.

## Go

Facendo clic sul pulsante *Go* il programma verrà eseguito normalmente fino a che termina o fino a che raggiunge un *punto di interruzione* (*breakpoint*). (I punti di interruzione sono descritti più avanti in questo capitolo.) Se avete finito il debug e volete che il programma continui normalmente, fate clic sul pulsante *Go*.

## Step

Facendo clic sul pulsante *Step* il debugger eseguirà la riga di codice successiva, quindi si fermerà nuovamente in attesa. L'elenco delle variabili globali e locali nella finestra *Debug Control* verrà aggiornato, se i loro valori cambiano. Se la riga successiva di codice è una chiamata di funzione, il debugger "entrerà" in quella funzione e salterà alla prima riga del codice della funzione.

## Over

Facendo clic sul pulsante *Over* verrà eseguita la riga successiva di codice, come con il pulsante *Step*. Se però la riga di codice successiva è una chiamata di funzione, con questo pulsante “passerete oltre” il codice della funzione. Il codice della funzione verrà eseguito a velocità normale, e il debugger si fermerà in attesa non appena la chiamata della funzione ritorna. Per esempio, se la riga di codice successiva è una chiamata a `print()`, non vi interessa quale codice si trovi all’interno della funzione predefinita `print()`; volette semplicemente che sullo schermo venga scritta la stringa che le avete passato. Per questo è più comune l’uso del pulsante *Over* che di *Step*.

## Out

Facendo clic sul pulsante *Out*, il debugger eseguirà le righe di codice a velocità normale fino a che non ritorna dalla funzione corrente. Se siete entrati in una chiamata di funzione con il pulsante *Step* e ora volette semplicemente continuare a eseguire le sue istruzioni fino alla fine, potete fare clic sul pulsante *Out* per uscire dalla chiamata di funzione corrente.

## Quit

Se volette fermare completamente la funzione di debug e non vi interessa continuare a eseguire il resto del programma in questa modalità, fate clic sul pulsante *Quit*, che farà immediatamente terminare il programma. Se volette a questo punto eseguire di nuovo normalmente il vostro programma, selezionate di nuovo **Debug > Debugger** per disattivare il debugger.

## Debug di un programma di addizione numerica

Aprite una nuova finestra di file editor e inserite il codice seguente:

```
print('Inserisci il primo numero da sommare.')
first = input()
print('Inserisci il secondo numero da sommare.')
second = input()
print('Inserisci il terzo numero da sommare.')
third = input()
print('La somma è ' + first + second + third)
```

Salvatelo con il nome *buggyAddingProgram.py* ed eseguitelo con il debugger disattivato. L’output del programma sarà una cosa di questo genere:

Inserisci il primo numero da sommare:

5

Inserisci il secondo numero da sommare:

3

Inserisci il terzo numero da sommare:

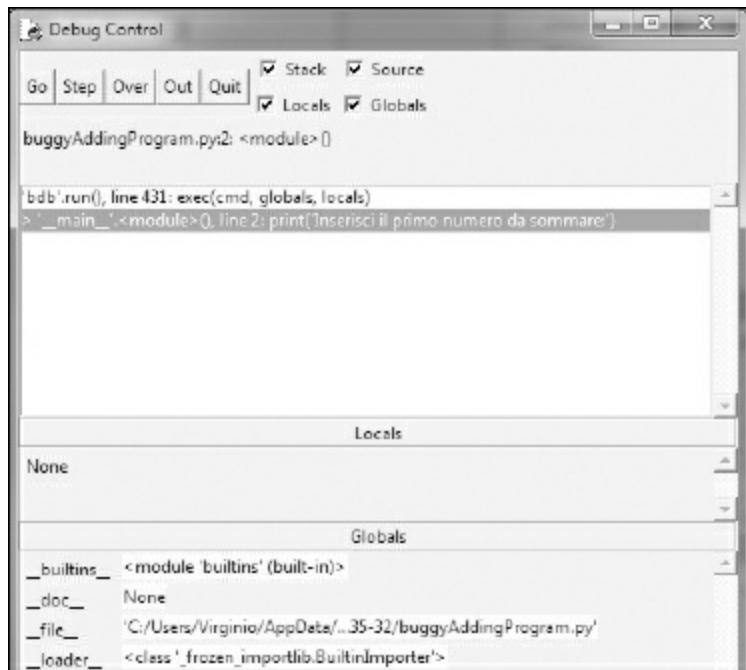
42

La somma è 5342

Il programma non è andato in crash, ma la somma è sbagliata. Attiviamo la finestra di *Debug Control* ed eseguiamo di nuovo il programma, questa volta sotto il debugger.

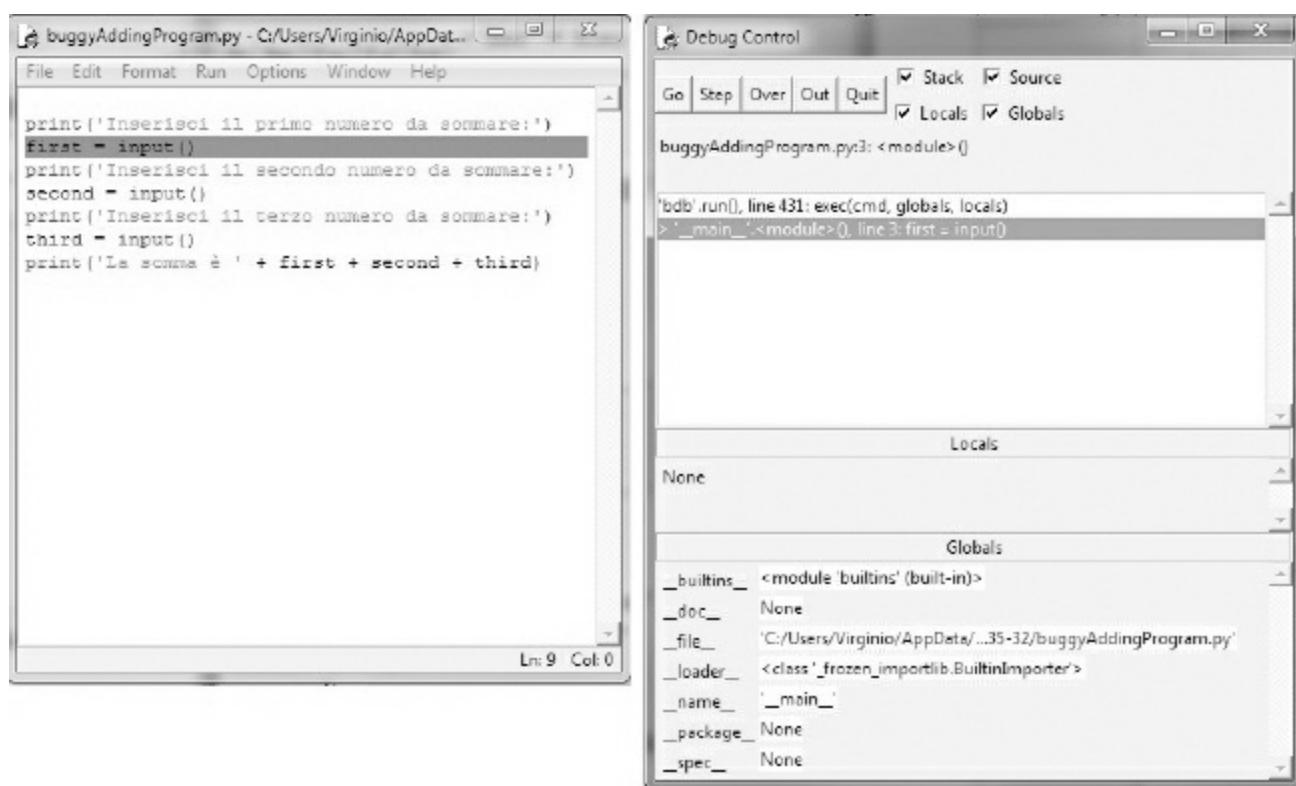
Quando premete F5 o selezionate **Run > Run Module** (con **Debug > Debugger** attivato e attivate

tutte le quattro caselle di controllo della finestra *Debug Control*), il programma inizia in uno stato di pausa sulla riga 1. Il debugger si ferma sempre sulla riga di codice che sta per eseguire e la finestra *Debug Control* si presenta come nella [Figura 10.2](#).



**Figura 10.2** - La finestra Debug Control quando il programma inizia sotto debugger.

Fate clic sul pulsante *Over* per eseguire la prima chiamata `print()`. Usate *Over*, in questo caso, anziché *Step*, perché non volete esaminare riga per riga il codice della funzione `print()`. La finestra *Debug Control* si aggiornerà alla riga 2 e nel file editor questa riga sarà evidenziata, come si vede nella [Figura 10.3](#). Questo vi fa vedere dove si trova in quel momento l'esecuzione del programma.



**Figura 10.3** - La finestra Debug Control dopo aver fatto clic su Over.

Fate clic nuovamente su *Over* per eseguire la chiamata di funzione `input()` e i pulsanti nella finestra IDLE si disattiveranno mentre IDLE aspetta che scriviate qualcosa in risposta alla chiamata `input()` nella finestra della shell interattiva. Inserite `5` e premete Invio. I pulsanti della finestra *Debug Control* torneranno a essere attivi.

Continuate a fare clic su *Over*, inserendo come numeri successivi `3` e `42`, fino a che il debugger non è sulla riga `7`, l'ultima chiamata di `print()` nel programma. La finestra *Debug Control* si presenterà come nella [Figura 10.4](#). Potete vedere nella sezione *Globals* che le variabili `first`, `second` e `third` sono impostate ai valori stringa `'5'`, `'3'` e `'42'`, anziché ai valori interi `5`, `3` e `42`. Quando l'ultima riga viene eseguita, queste stringhe vengono concatenate anziché sommate, provocando l'errore.



**Figura 10.4** - La finestra Debug Control sull'ultima riga. La variabili sono impostate a stringhe, ed è questo che provoca l'errore.

Procedere nel programma passo per passo con il debugger è utile, ma può essere anche un procedimento molto lento. Spesso si vuole che il programma venga eseguito normalmente fino a che non raggiunge una certa riga di codice. Potete configurare il debugger perché si comporti in questo modo utilizzando i punti di interruzione.

## Punti di interruzione

Si può definire un **punto di interruzione** o **breakpoint** su una specifica riga di codice, il che costringe il debugger a fermarsi in pausa ogni volta che l'esecuzione del programma raggiunge quella riga. Aprite una nuova finestra di file editor e inserite il programma seguente, che simula il lancio di una moneta per mille volte. Salvatelo con il nome `coinFlip.py`.

```

import random
heads = 0
for i in range(1, 1001):
    if random.randint(0, 1) == 1:
        heads = heads + 1
    if i == 500:
        print('Siamo a metà!')
print('Testa è uscito ' + str(heads) + ' volte.')

```

La chiamata a `random.randint(0, 1)` ① restituirà 0 circa metà delle volte e 1 negli altri casi. Potete utilizzare questo programma per simulare il lancio di una moneta non truccata, dove 1 rappresenta testa. Se eseguite il programma senza il debugger, rapidamente produce qualcosa di simile a questo:

Siamo a metà!  
Testa è uscito 493 volte.

Se eseguite il programma sotto il debugger, dovrete fare clic sul pulsante *Over* per migliaia di volte prima che il programma termini. Se foste interessati al valore di `heads` a metà dell'esecuzione del programma, quando la moneta è stata lanciata 500 volte, potreste invece impostare un punto di interruzione sulla riga `print('Siamo a metà!')` ②. Per impostare un punto di interruzione, fate clic destro sulla riga nel file editor e selezionate **Set Breakpoint**, come nella [Figura 10.5](#).



**Figura 10.5** - Impostazione di un punto di interruzione.

Non è il caso di impostare un punto di interruzione sulla riga dell'enunciato `if`, poiché questo viene eseguito a ogni singola iterazione del ciclo. Impostando il punto di interruzione nel codice dell'enunciato `if`, il debugger si interrompe solo quando l'esecuzione entra nella clausola `if`.

La riga con il punto di interruzione verrà evidenziata in giallo nel file editor. Quando eseguite il programma sotto il debugger, inizierà in uno stato di pausa sulla prima riga, come al solito, ma se fate clic su *Go*, il programma girerà normalmente fino a che non raggiunge la riga in cui è stato impostato il punto di interruzione. A quel punto potete fare clic su *Go*, *Over*, *Step* o *Out* per continuare normalmente.

Se volete eliminare un punto di interruzione, fate clic destro sulla riga nel file editor e selezionate dal menu **Clear Breakpoint**. L'evidenziazione in giallo scomparirà e in futuro il debugger non si interromperà più su quella riga.

# Riepilogo

Asserzioni, eccezioni, logging e debugger sono tutti strumenti preziosi per identificare e prevenire gli errori nei programmi. Le **asserzioni** con l'enunciato assert di Python sono un buon modo per mettere in atto “controlli di sicurezza” che generano avvertimenti precoci quando una condizione necessaria non è vera. Le asserzioni sono solo per errori da cui il programma non deve tentare di riprendersi e devono entrare in azione rapidamente; altrimenti, dovete sollevare un’eccezione.

Un’**eccezione** può essere catturata e gestita con gli enunciati try ed except. Il modulo logging è un buon modo per analizzare il codice mentre è in esecuzione ed è molto più comodo da usare rispetto alla funzione print(), grazie ai suoi diversi livelli e alla possibilità di scrivere il log in un file di testo.

Il **debugger** consente di percorrere il programma una riga alla volta. In alternativa, potete eseguire il programma a velocità normale e fare in modo che il debugger metta in pausa l’esecuzione ogni volta che raggiunge una riga su cui è stato impostato un **punto di interruzione**. Utilizzando il debugger, potete vedere lo stato del valore di qualsiasi variabile in qualsiasi momento nell’arco della vita del programma.

Questi strumenti e queste tecniche di debugging vi aiuteranno a scrivere programmi che funzionano. L’introduzione involontaria di errori nel codice è una cosa che capita a tutti, non importa quanti anni di esperienza abbiate nella programmazione.

## Domande di ripasso

1. Scrivete un enunciato assert che provochi un AssertionError se la variabile spam è un intero minore di 10.
2. Scrivete un enunciato assert che provochi un AssertionError se le variabili eggs e bacon contengono stringhe identiche, anche se scritte con diverse combinazioni di maiuscole e minuscole (in altre parole: 'hello' ed 'hello' sono considerate identiche, e lo stesso dicasì per 'arrivederci' e 'ARRIvederci').
3. Scrivete un enunciato assert che provochi sempre un AssertionError.
4. Quali sono le due righe che il vostro programma deve possedere, per poter chiamare logging.debug()?
5. Quali sono le due righe che il vostro programma deve possedere per poter fare inviare a logging.debug() un messaggio di logging a un file *programLog.txt*?
6. Quali sono i cinque livelli di logging?
7. Quale riga di codice potete inserire per disattivare tutti i messaggi di logging nel vostro programma?
8. Perché l’uso dei messaggi di logging è una scelta migliore che l’uso di print() per visualizzare lo stesso messaggio?
9. Quali sono le differenze fra i pulsanti *Step*, *Over* e *Out* nella finestra del debugger *Debug Control*?
10. Se fate clic su *Go* nella finestra *Debug Control*, quando si fermerà il debugger?
11. Che cos’è un punto di interruzione o breakpoint?
12. Come si imposta un punto di interruzione su una riga di codice in IDLE?

## Un po’ di pratica

Per esercitarvi, scrivete un programma che svolge l’attività seguente.

## Debug del lancio di una moneta

Il programma seguente vuole essere un semplice gioco in cui si deve indovinare il risultato del lancio di una moneta. Il giocatore può fare due congetture (è un gioco facile). Il programma però contiene parecchi errori. Eseguite il programma un po' di volte, per trovare gli errori che non lo fanno funzionare correttamente.

```
import random
guess = ''
while guess not in ('testa', 'croce'):
    print('Indovina l\'esito del lancio! Scrivi testa o croce:')
    guess = input()
toss = random.randint(0, 1) # 0 è croce, 1 è testa
if toss == guess:
    print('Indovinato!')
else:
    print('Sbagliato! Prova ancora!')
    guesss = input()
    if toss == guess:
        print('Indovinato!')
    else:
        print('Niente da fare. Proprio non sei portato per questo gioco.')
```

# Web scraping

In quei rari, terrificanti momenti in cui sono senza Wi-Fi, mi rendo conto che quel che faccio con il mio computer è in realtà quello che faccio in **Internet**. Per pura abitudine mi ritrovo a cercare di controllare la **posta elettronica**, a leggere i **tweet** degli amici o a rispondere alla domanda “Kurtwood Smith ha ricoperto ruoli importanti prima di recitare nel RoboCop originale del 1987?”<sup>1</sup>.

Dato che una parte così importante del lavorare con un computer comporta andare in Internet, sarebbe splendido se i vostri programmi potessero andare online. **Web scraping** è il termine che indica l’uso di un programma per **scaricare ed elaborare contenuti dal Web**. Per esempio, Google gestisce molti programmi di web scraping per indicizzare le pagine web per il suo motore di ricerca. In questo capitolo, vedremo vari moduli che semplificano il lavoro con le pagine web in Python.

- **webbrowser** È fornito con Python e apre un browser a una pagina specifica.
- **Requests** Scarica file e pagine web.
- **Beautiful Soup** Analizza l’HTML, il formato in cui sono scritte le pagine web.
- **Selenium** Lancia e controlla un browser web. Selenium è in grado di compilare moduli e di simulare clic del mouse in questo browser.

## Progetto: mapIt.py con il modulo webbrowser

La funzione `open()` del modulo `webbrowser` può lanciare un **browser** aprendo un **URL** specificato. Inserite quanto segue nella shell interattiva:

```
>>> import webbrowser
>>> webbrowser.open('http://inventwithpython.com/')
```

Si aprirà una scheda del web browser, all’URL <http://inventwithpython.com>. Questa è più o meno l’unica cosa che il modulo `webbrowser` può fare. Anche così, la funzione `open()` rende possibili alcune

attività interessanti. Per esempio, è noioso copiare un indirizzo stradale negli Appunti e poi trovarlo su una mappa di Google Maps. Potete eliminare qualche passaggio scrivendo un semplice script per lanciare automaticamente la mappa nel browser utilizzando i contenuti degli Appunti. In questo modo dovete solo copiare l'indirizzo negli Appunti ed eseguire lo script, e la mappa verrà caricata per voi. Questo è ciò che fa il programma:

- prende un indirizzo stradale dagli argomenti della riga di comando o dagli Appunti;
- apre il browser web sulla pagina di Google Maps per quell'indirizzo.

Questo significa che il vostro codice deve fare le cose seguenti:

- leggere gli argomenti della riga di comando da `sys.argv`;
- leggere i contenuti degli Appunti;
- chiamare la funzione `webbrowser.open()` per aprire il browser web.

Aprite una nuova finestra di file editor e salvate con il nome *mapIt.py*.

## Passo 1: stabilire l'URL

Sulla base delle istruzioni riportate nell'[Appendice B](#), impostate *mapIt.py* in modo che quando lo lanciate dalla riga di comando, per esempio così...

```
C:\> mapit 870 Valencia St, San Francisco, CA 94110
```

... lo script usi gli argomenti da riga di comando invece degli Appunti. Se non ci sono argomenti nella riga di comando, allora il programma saprà di dover usare i contenuti degli Appunti.

In primo luogo, dovete stabilire quale URL utilizzare per un dato indirizzo stradale. Se caricate <http://maps.google.com/> nel browser e cercate un indirizzo l'URL nella barra degli indirizzi è simile a questo:

<https://www.google.com/maps/place/870+Valencia+St/@37.7590311,-122.4215096,17z/data=!3m1>

L'indirizzo è nell'URL, ma vi si trova anche molto altro testo. I siti web spesso aggiungono agli URL dati ulteriori, per poter tracciare i visitatori o per personalizzare i loro siti. Se provate semplicemente a inserire

<https://www.google.com/maps/place/870+Valencia+St+San+Francisco+CA/>, scoprirete che comunque si apre la pagina giusta. Il vostro programma quindi può essere impostato in modo da aprire un browser web sulla pagina '[www.google.com/maps/place/stringa\\_indirizzo](http://www.google.com/maps/place/stringa_indirizzo)' (dove *stringa\_indirizzo* è l'indirizzo stradale che volete visualizzare sulla mappa).

## Passo 2: gestire gli argomenti da riga di comando

Iniziate a scrivere il vostro codice in questo modo:

```
#! python3
# mapIt.py - Lancia una mappa nel browser utilizzando un indirizzo
# ricavato dalla riga di comando o dagli Appunti.

import webbrowser, sys
if len(sys.argv) > 1:
    # Prende l'indirizzo dalla riga di comando.
    address = ' '.join(sys.argv[1:])

# DA FARE: Prendere l'indirizzo dagli Appunti.
```

Dopo la riga `#!` iniziale, dovete importare il modulo `webbrowser` per lanciare il browser e importare il modulo `sys` per leggere i potenziali argomenti dalla riga di comando. La variabile `sys.argv` memorizza una lista con il nome di file del programma e gli argomenti da riga di comando. Se la lista contiene qualcosa oltre al nome del file, `len(sys.argv)` viene valutata a un intero maggiore di 1, il che significa che sono stati forniti argomenti da riga di comando.

Gli argomenti da riga di comando di solito sono separati da spazi ma, in questo caso, volete interpretare tutti gli argomenti come un'unica stringa. Dato che `sys.argv` è una lista di stringhe, potete passarla al metodo `join()`, che restituisce un unico valore stringa. In questa stringa non deve esserci il nome del programma, perciò, anziché `sys.argv`, dovete passare `sys.argv[1:]` in modo da escludere il primo elemento. La stringa finale a cui viene valutata questa espressione è memorizzata nella variabile `address`.

Se eseguite il programma inserendo nella riga di comando

```
mapit 879 Valencia St, San Francisco, CA 94110
```

la variabile `sys.argv` conterrà questo valore lista:

```
['mapIt.py', '870', 'Valencia', 'St', 'San', 'Francisco', 'CA', '94110']
```

La variabile `address` conterrà la stringa '870 Valencia St, San Francisco, CA 94110'.

## Passo 3: gestire i contenuti degli Appunti e lanciare il browser

Modificate il vostro codice in questo modo:

```
#! python3
# mapIt.py - Lancia una mappa nel browser utilizzando un indirizzo
# ricavato dalla riga di comando o dagli Appunti.

import webbrowser, sys, pyperclip
if len(sys.argv) > 1:
    # Prende l'indirizzo dalla riga di comando.
    address = ' '.join(sys.argv[1:])
else:
    # Prende l'indirizzo dagli Appunti.
    address = pyperclip.paste()

webbrowser.open('https://www.google.com/maps/place/' + address)
```

Se non vi sono argomenti da riga di comando, il programma assumerà che l'indirizzo sia memorizzato

negli Appunti.

Potete ottenere i contenuti presenti negli Appunti con `pyperclip.paste()` e memorizzarli in una variabile `address`.

Infine, per lanciare un browser e aprirlo all'URL di Google Maps, chiamate la funzione `webbrowser.open()`.

Alcuni dei programmi che scriverete svolgeranno compiti enormi che vi faranno risparmiare ore, ma può dare altrettanta soddisfazione usare un programma che vi fa comodamente risparmiare qualche secondo ogni volta che svolgete una attività comune, come trovare la mappa per un indirizzo.

La [Tabella 11.1](#) mette a confronto i passi necessari per visualizzare una mappa con e senza `mapIt.py`.

**Tabella 11.1** - Visualizzare una mappa con e senza `mapIt.py`.

Ottenerne una mappa manualmente	Con <code>mapIt.py</code>
Evidenziare l'indirizzo. Copiare l'indirizzo Aprire il browser web. Andare a <a href="http://maps.google.com/">http://maps.google.com/</a> . Fare clic sul campo di testo per l'indirizzo. Incollare l'indirizzo. Premere Invio.	Evidenziare l'indirizzo. Copiare l'indirizzo. Eseguire <code>mapIt.py</code> .

Vedete come `mapIt.py` renda meno noioso questo compito?

## Idee per programmi simili

Purché abbiate un URL, il modulo `webbrowser` permette di eliminare i passi necessari per aprire il browser e andare a un sito web. Altri programmi potrebbero usare questa funzione per fare altre cose:

- aprire tutti i link presenti in una pagina in schede diverse del browser;
- aprire il browser all'URL del servizio meteorologico locale;
- aprire vari siti di social networking che frequentate regolarmente.

## Scaricare file dal Web con il modulo `requests`

Il modulo `requests` permette di scaricare facilmente file dal Web senza doversi preoccupare di questioni complicate come gli errori di rete, i problemi di connessione e la compressione dei dati. Il modulo `requests` non è fornito insieme a Python, perciò dovete prima installarlo. Dalla riga di comando, eseguite `pip install requests`. (Nell'[Appendice A](#) trovate ulteriori particolari su come installare moduli di terze parti.)

Il modulo `requests` è stato scritto perché il modulo `urllib2` di Python è troppo complicato da usare. In effetti, prendete una penna indeleibile e cancellate tutto questo paragrafo. Dimenticatevi di aver mai sentito parlare di `urllib2`. Se dovete scaricare qualcosa dal Web, usate il modulo `requests`.

Poi, fate un semplice test per verificare che il modulo `requests` sia stato installato correttamente. Inserite nella shell interattiva:

```
>>> import requests
```

Se non compaiono messaggi d'errore, requests è stato installato perfettamente.

## Scaricare una pagina web con la funzione requests.get()

La funzione `requests.get()` prende una stringa che contiene un URL da scaricare. Chiamando `type()` sul valore di ritorno di `requests.get()`, potete vedere che restituisce un oggetto `Response` che contiene la risposta data dal server web alla vostra richiesta. Spiegherò meglio in seguito l'oggetto `Response`, ma per il momento inserite quanto segue nella shell interattiva, con il computer collegato a Internet:

```
❶ >>> import requests
❷ >>> res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
❸ >>> type(res)
<class 'requests.models.Response'>
❹ >>> res.status_code == requests.codes.ok
True
>>> len(res.text)
178981
>>> print(res.text[:250])
```

The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare

This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or reuse it under the terms of the Project Gutenberg License.

L'URL punta a una pagina web di testo in cui si trova tutto il testo inglese di *Romeo and Juliet* di Shakespeare, nel sito dedicato a questo libro ❶. Potete stabilire che la richiesta di questa pagina web ha avuto successo controllando l'attributo `status_code` dell'oggetto `Response`. Se è uguale al valore di `requests.codes.ok`, allora tutto è andato liscio ❷. (Incidentalmente, il codice di status per "OK" nel protocollo HTTP è 200. Sicuramente conoscete già un altro codice di stato, il 404 che sta per "non trovato", "Not Found".)

Se la richiesta è andata a buon fine, la pagina web scaricata è memorizzata come stringa nella variabile `text` dell'oggetto `Response`. Questa variabile contiene una lunga stringa con tutto il testo dell'opera; la chiamata a `len(res.text)` mostra che è lunga più di 178.000 caratteri. Infine, chiamando `print(res.text[:250])` vengono visualizzati solo i primi 250 caratteri.

## Controllare se ci sono errori

Come avete visto, l'oggetto `Response` ha un attributo `status_code` che può essere confrontato con `requests.codes.ok` per vedere se il download ha avuto successo. Un procedimento più semplice per verificare se tutto è andato a buon fine è chiamare il metodo `raise_for_status()` sull'oggetto `Response`. Questo solleverà un'eccezione se vi è stato un errore nello scaricamento del file e non farà nulla invece se è andato a buon fine. Inserite quanto segue nella shell interattiva:

```
>>> res = requests.get('http://inventwithpython.com/page_that_does_not_exist')
>>> res.raise_for_status()
Traceback (most recent call last):
  File "<pyshell#138>", line 1, in <module>
    res.raise_for_status()
  File "C:\Python34\lib\site-packages\requests\models.py", line 773, in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 404 Client Error: Not Found
```

Il metodo `raise_for_status()` è un buon modo per essere sicuri che un programma si fermi se si verificano problemi di scaricamento. È una buona cosa: volete che il vostro programma si fermi nel caso in cui si verifichi qualche errore inatteso. Se invece il fallimento del download non è motivo di interruzione per il vostro programma, potete racchiudere la riga `raise_for_status()` fra enunciati `try` ed `except` per gestire l'errore senza provocare un crash.

```
import requests
res = requests.get('http://inventwithpython.com/page_that_does_not_exist')
try:
    res.raise_for_status()
except Exception as exc:
    print('Si è verificato un problema: %s' % (exc))
```

Questa chiamata al metodo `raise_for_status()` fa sì che il programma produca:

```
Si è verificato un problema: 404 Client Error: Not Found
```

Chiamate sempre `raise_for_status()` dopo aver chiamato `requests.get()`. Assicuratevi sempre che il download abbia funzionato, prima che il programma continui.

## Salvare sul disco fisso i file scaricati

Da qui, potete **salvare la pagina web in un file** sul vostro disco fisso con la normale funzione `open()` e il metodo `write()`. Vi sono alcune piccole differenze, però. In primo luogo, dovete aprire il file in modalità **scrittura binaria**, passando la stringa '`wb`' come secondo argomento a `open()`.

Anche se la pagina è in puro testo (come per *Romeo and Juliet* che avete scaricato prima), dovete scrivere dati binari anziché dati testo per mantenere la **codifica Unicode** del testo.

### Codifiche Unicode

Le codifiche Unicode vanno al di là degli scopi di questo libro, ma potete scoprire di più in proposito visitando queste pagine web:

- Joel on Software: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!): <http://www.joelonsoftware.com/articles/Unicode.html>.
- Pragmatic Unicode: <http://nedbatchelder.com/text/unipain.html>.

Per scrivere la pagina web in un file, potete usare un ciclo `for` con il metodo `iter_content()` dell'oggetto Response.

```

>>> import requests
>>> res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
>>> res.raise_for_status()
>>> playFile = open('RomeoAndJuliet.txt', 'wb')
>>> for chunk in res.iter_content(100000):
    playFile.write(chunk)

100000
78981
>>> playFile.close()

```

Il metodo `iter_content()` restituisce “pezzi” di contenuto a ogni iterazione nel ciclo. Ciascun pezzo è del tipo di dati `bytes` e dovete specificare quanti byte conterrà. Una buona dimensione sono centomila byte, perciò passate 100000 come argomento a `iter_content()`.

Il file *RomeoAndJuliet.txt* ora si trova nella directory di lavoro corrente. Notate che, anche se il nome del file sul sito web era *pg1112.txt*, il file sul vostro disco fisso ha un nome diverso. Il modulo `requests` gestisce solo lo scaricamento dei contenuti delle pagine web; una volta che la pagina è scaricata, si tratta semplicemente di dati nel vostro programma. Anche se dovesse cadere la connessione a Internet dopo lo scaricamento, tutti i dati della pagina sarebbero ancora sul vostro computer.

Il metodo `write()` restituisce il numero dei byte scritti nel file. Nell'esempio, c'erano 100.000 byte nel primo pezzo, e la parte restante del file richiedeva solo 78.981 byte.

Per riepilogare, ecco il processo completo per scaricare e salvare un file.

1. Chiamate `requests.get()` per scaricare il file.
2. Chiamate `open()` con argomento `'wb'` per creare un nuovo file in modalità scrittura binaria.
3. Ciclate sul metodo `iter_content()` dell'oggetto `Response`.
4. Chiamate `write()` in ciascuna iterazione per scrivere i contenuti su file.
5. Chiamate `close()` per chiudere il file.

E questo è tutto per il modulo `requests`. Il ciclo `for` e il metodo `iter_content()` possono sembrare complicati rispetto al flusso `open()`, `write()`, `close()` che avete usato per scrivere file di testo, ma serve per essere sicuri che il modulo `requests` non consumi troppa memoria, anche se si scaricano file molto grandi. Potete scoprire le altre caratteristiche del modulo `requests` all'indirizzo <http://requests.readthedocs.org/>.

## HTML

Prima di smontare pagine web, dovete imparare un po' di HTML. Dovrete anche vedere come accedere ai potenti strumenti di sviluppo del vostro browser, che renderanno molto più facile estrarre informazioni dal Web.

## Risorse per imparare HTML

L'*HyperText Markup Language* (HTML) è il formato in cui sono scritte le pagine web. Qui si dà per scontato che abbiate un minimo di esperienza con questo linguaggio, ma, se avete bisogno di un tutorial per iniziare a studiarlo, vi consiglio uno di questi siti:

- <http://htmldog.com/guides/html/beginner/>.
- <http://www.codecademy.com/tracks/web/>.

- <https://developer.mozilla.org/en-US/learn/html/>.

## Un ripasso rapido

Nel caso sia passato un po' di tempo da quando avete usato HTML, ecco un breve ripasso degli elementi fondamentali. Un file HTML è un file di puro testo con estensione *.html*. Il testo in questi file è racchiuso fra **tag (marcatori)**, che sono parole fra parentesi angolari. Il tag dice al browser come **formattare** la pagina web; un tag di apertura e uno di chiusura racchiudono del testo e l'insieme forma un **elemento**. Il *testo* è il contenuto fra tag di apertura e di chiusura. Per esempio, il seguente HTML visualizzerà *Ciao mondo!* nel browser, con *Ciao* in grassetto, come nella [Figura 11.1](#):



**Figura 11.1** - Ciao mondo! visualizzato nel browser.

```
<strong>Hello</strong> mondo!
```

Il tag `<strong>` di apertura dice che il testo seguente sarà visualizzato in grassetto. Il tag di chiusura `</strong>` dice al browser dove finisce il testo in grassetto.

HTML consente molti tag diversi, alcuni dei quali hanno proprietà aggiuntive sotto forma di attributi che vengono indicati all'interno delle parentesi angolari. Per esempio, il tag `<a>` identifica testo che deve essere un link. L'URL a cui il testo rimanda è determinato dall'attributo `href`. Ecco un esempio:

```
All's free <a href="http://inventwithpython.com">Python books</a>.
```

In un browser il risultato sarà come quello che si vede nella [Figura 11.2](#).



**Figura 11.2** - Un link visualizzato nel browser.

Alcuni elementi hanno un attributo `id` che viene utilizzato per identificare in modo univoco quell'elemento nella pagina. Spesso si dice al programma di cercare un elemento in base al suo attributo `id`, perciò stabilire quale sia l'attributo `id` di un elemento mediante gli strumenti per gli sviluppatori messi a disposizione dal browser è un compito comune nella scrittura di programmi di

## Vedere l'HTML sorgente di una pagina web

Avrete bisogno di **esaminare il sorgente HTML** delle pagine web con cui lavoreranno i vostri programmi. Per poterlo fare, fate un clic destro (o Ctrl-clic sotto OS X) su una pagina nel vostro browser e selezionate *Visualizza sorgente* o *Visualizza sorgente pagina*, per vedere il testo HTML della pagina ([Figura 11.3](#)). Quello è il testo che il vostro browser riceve effettivamente: il browser sa come visualizzare (o *rendere*) la pagina web a partire da quell'HTML.



**Figura 11.3** - Come vedere il sorgente di una pagina web.

Vi consiglio di esaminare il sorgente HTML di qualcuno dei siti che preferite. Non è importante se non capite del tutto quello che vi si presenta: non vi servirà una completa padronanza dell'HTML per scrivere semplici programmi di web scraping – in fin dei conti, non dovrete costruire i vostri siti web. Vi serve capire quel tanto che basta per estrarre dati da un sito esistente.

## Aprire gli strumenti per gli sviluppatori del browser

Oltre a vedere il codice sorgente di una pagina web, potete esaminare il codice HTML utilizzando gli strumenti per gli sviluppatori messi a disposizione del browser. In Chrome, Internet Explorer e Edge per Windows, gli strumenti per gli sviluppatori sono già installati e potete semplicemente premere F12 per farli comparire ([Figura 11.4](#)). Premendo nuovamente F12, gli strumenti scompariranno.



**Figura 11.4** - La finestra degli Strumenti per gli sviluppatori nel browser Chrome.

In Chrome, potete far comparire gli strumenti per gli sviluppatori anche selezionando **Altri strumenti > Strumenti per sviluppatori** (o premendo Ctrl-Maiusc-I). In OS X, gli Strumenti per gli sviluppatori di Chrome si aprono premendo **⌘-opzione-I**.

In Firefox, gli strumenti possono essere chiamati premendo Ctrl-Maiusc-C in Windows e Linux, oppure premendo **⌘-opzione-C** in OS X. La visualizzazione è molto simile a quella degli strumenti per gli sviluppatori di Chrome.

In Safari, aprite la finestra *Preferenze* e nel pannello *Avanzate* attivate l'opzione *Mostra il menu di sviluppo nella barra dei menu*. Una volta abilitata questa opzione, potete chiamare gli strumenti premendo **⌘-opzione-I**.

Dopo aver abilitato o installato gli strumenti per gli sviluppatori nel vostro browser, potete fare un clic destro su qualsiasi parte della pagina web e selezionare *Ispeziona elemento* dal menu di scelta rapida, per visualizzare il codice HTML a cui si deve quella parte della pagina, il che può essere utile quando inizierete ad analizzare l'HTML per i vostri programmi di web scraping.

## Non utilizzate le espressioni regolari per analizzare HTML

Identificare un brano specifico di HTML in una stringa sembrerebbe un compito perfetto per le espressioni regolari, ma vi consiglio di non seguire questa strada. Vi sono molti modi diversi in cui il codice HTML può essere formattato in modo valido, ma cercare di

catturare in un'espressione regolare tutte le varianti possibili potrebbe rivelarsi noioso e indurre facilmente in errore.

È meno probabile che provochi errori un modulo sviluppato specificamente per l'analisi di HTML, come Beautiful Soup.

Potete trovare un'analisi più estesa del perché non analizzare HTML con le espressioni regolari all'indirizzo

<http://stackoverflow.com/a/1732454/1893164/>.

## Uso degli strumenti per sviluppatori per trovare elementi HTML

Una volta che il vostro programma ha scaricato una pagina web utilizzando il modulo `requests`, avrete i contenuti HTML della pagina in un unico valore stringa. A questo punto dovete stabilire quale parte dell'HTML corrisponda alle informazioni a cui siete interessati, e qui possono venire in aiuto gli strumenti per gli sviluppatori messi a disposizione dal browser.

Supponiamo vogliate scrivere un programma per estrarre dati sulle previsioni del tempo dal sito <http://weather.gov/>.

Prima di scrivere una riga di codice, effettuate qualche ricerca. Se visitate il sito e cercate il codice postale 94105, il sito vi porterà a una pagina che mostra le previsioni del tempo per quella zona.

E se foste interessati a estrarre le informazioni sulla temperatura per quel codice postale? Fate clic destro dove si trova sulla pagina (o Control-clic sotto OS X) e selezionate *Ispeziona* dal menu di scelta rapida. Questo farà aprire la finestra con gli Strumenti per gli sviluppatori, che vi mostrerà l'HTML che produce quella particolare parte della pagina web.

La Figura 11,5 mostra gli strumenti per gli sviluppatori aperti sul codice HTML della temperatura.



Figura 11,5 - Ispezionare l'elemento che contiene il testo relativo alla temperatura con gli strumenti per gli sviluppatori.

Dagli strumenti per sviluppatori, potete vedere che l'HTML responsabile della parte relativa alla temperatura nella pagina web è `<p class="myforecast-current-lrg">56°F</p>`. È esattamente quello che

cercavate. Sembra che le informazioni sulla temperatura siano contenute all'interno di un elemento <p> con classe myforecast-current-lrg. Ora che sapete che cosa cercare, il modulo BeautifulSoup vi aiuterà a trovarlo all'interno della stringa.

## Analizzare HTML con il modulo BeautifulSoup

Beautiful Soup è un modulo per **estrarre informazioni da una pagina HTML** (ed è molto meglio, a questo scopo, delle espressioni regolari). Il nome del modulo BeautifulSoup è bs4 (che sta per Beautiful Soup, versione 4). Per installarlo, dovete eseguire pip install beautifulsoup4 dalla riga di comando. (Consultate l'[Appendice A](#) per le istruzioni sull'installazione di moduli di terze parti.) Mentre beautifulsoup4 è il nome da usare l'installazione, per importare questo modulo si deve eseguire import bs4. Per questo capitolo, gli esempi di Beautiful Soup analizzeranno e identificheranno le parti di un file HTML (è quello che si chiama anche **parsing** del file) che si trova sul disco fisso. Aprite una nuova finestra di file editor in IDLE, inserite quanto segue e salvate il file con il nome *example.html*. In alternativa, potete scaricarne la versione inglese dall'indirizzo <http://nostarch.com/automatestuff/>.

```
<!-- Questo è il file di esempio example.html. -->

<html><head><title>Il titolo del sito web</title></head>
<body>
<p>Scaricate il mio libro su <strong>Python</strong> dal mio sito <a href="http://inventwithpython.com"></a>.</p>
<p class="slogan">Imparare Python non è mai stato così facile!</p>
<p>By <span id="author">Al Sweigart</span></p>
</body></html>
```

Come potete vedere, anche un semplice file HTML comporta molti tag e molti attributi, e nei siti web complessi le cose si fanno rapidamente confuse. Per fortuna, Beautiful Soup rende molto più facile lavorare con HTML.

## Creare un oggetto BeautifulSoup da HTML

La funzione bs4.BeautifulSoup() deve essere chiamata con una stringa contenente l'HTML che dovrà analizzare.

La funzione restituisce un oggetto BeautifulSoup. Inserite quanto segue nella shell interattiva mentre il vostro computer è collegato a Internet:

```
>>> import requests, bs4
>>> res = requests.get('http://nostarch.com')
>>> res.raise_for_status()
>>> noStarchSoup = bs4.BeautifulSoup(res.text)
>>> type(noStarchSoup)
<class 'bs4.BeautifulSoup'>
```

Questo codice usa requests.get() per scaricare la pagina principale dal sito web di No Starch Press e poi passa l'attributo `text` della risposta a `bs4.BeautifulSoup()`. L'oggetto BeautifulSoup che restituisce è memorizzato in una variabile con nome `NoStarchSoup`. Potete anche caricare un file HTML dal disco fisso passando a `bs4.BeautifulSoup()` un oggetto File. Inserite quanto segue nella shell interattiva (assicuratevi prima che il file *example.html* si trovi nella directory di lavoro):

```
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile)
>>> type(exampleSoup)
<class 'bs4.BeautifulSoup'>
```

Una volta che avete un oggetto BeautifulSoup, potete usare i suoi metodi per trovare parti specifiche di un documento HTML.

## Trovare un elemento con il metodo select()

Potete recuperare un elemento di una pagina web da un oggetto BeautifulSoup chiamando il metodo `select()` e passandogli una stringa contenente un selettore CSS per l'elemento che state cercando. I selettori sono come espressioni regolari: specificano uno schema da cercare, in questo caso in pagine HTML invece che in generiche stringhe di testo. Una analisi completa della sintassi dei selettori CSS va oltre gli obiettivi di questo libro (trovate un buon tutorial fra le risorse nel sito <http://nostarch.com/automatestuff/>); questa è solo una breve introduzione. La [Tabella 11.2](#) mostra alcuni esempi degli schemi più comuni per i selettori CSS.

**Tabella 11.2** - Esempi di selettori CSS.

Selettore passato al metodo <code>select()</code>	Identificherà...
<code>soup.select('div')</code>	Tutti gli elementi <div>
<code>soup.select('#author')</code>	L'elemento con un attributo id author
<code>soup.select('.notice')</code>	Tutti gli elementi che usano un attributo CSS class chiamato notice
<code>soup.select('div span')</code>	Tutti gli elementi <span> che si trovano entro un elemento <div>
<code>soup.select('div &gt; span')</code>	Tutti gli elementi <span> che sono direttamente all'interno di un elemento <div>, senza alcun altro elemento fra l'uno e l'altro
<code>soup.select('input[name]')</code>	Tutti gli elementi <input> che hanno un attributo name con qualsiasi valori
<code>soup.select('input[type="button"]')</code>	Tutti gli elementi <input> che hanno un attributo type con valore button.

I vari schemi si possono combinare per effettuare ricerche più complesse. Per esempio, `soup.select('p #author')` troverà qualsiasi elemento che abbia un attributo id author, purché sia anche all'interno di un elemento <p>.

Il metodo `select()` restituirà una lista di oggetti Tag, che è il modo in cui Beautiful Soup rappresenta un elemento HTML. La lista conterrà un oggetto Tag per ogni occorrenza identificata nell'HTML dell'oggetto BeautifulSoup. I valori dei tag possono essere passati alla funzione `str()` per vedere i tag HTML che rappresentano.

I valori Tag possono avere anche un attributo `attrs` che mostra tutti gli attributi HTML del tag sotto forma di dizionario. Utilizzando il file *example.html* visto prima, inserite quanto segue nella shell interattiva.

```
>>> import bs4
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile.read())
```

```
>>> elems = exampleSoup.select('#author')
>>> type(elems)
<class 'list'>
>>> len(elems)
1
>>> type(elems[0])
<class 'bs4.element.Tag'>
>>> elems[0].getText()
'Al Sweigart'
>>> str(elems[0])
'<span id="author">Al Sweigart</span>'
>>> elems[0].attrs
{'id': 'author'}
```

Questo codice estrae l'elemento con `id="author"` dal nostro esempio di HTML. Usiamo `select('#author')` per restituire una lista di tutti gli elementi con `id="author"`. Memorizziamo questa lista di oggetti Tag nella variabile `elems`, e `len(elems)` ci dice che esiste un solo oggetto Tag nella lista: è stata identificata un'unica occorrenza. Chiamando `getText()` sull'elemento si ha di ritorno il testo dell'elemento, cioè il contenuto fra tag di apertura e chiusura, in questo caso, 'Al Sweigart'.

Se si passa l'elemento a `str()` si ha di ritorno una stringa con i tag di apertura e chiusura e il testo dell'elemento. Infine, `attrs` ci dà un dizionario con l'attributo dell'elemento, 'id', e il valore dell'attributo `id`, cioè 'author'.

Potete estrarre dall'oggetto BeautifulSoup anche tutti gli elementi `<p>`. Inserite quanto segue nella shell interattiva:

```
>>> pElems = exampleSoup.select('p')
>>> str(pElems[0])
'<p>Scaricate il mio libro su <strong>Python</strong> dal mio sito <a href="http://inventwithpython.com"></a>.</p>'
>>> pElems[0].getText()
'Scaricate il mio libro su Python dal mio sito.'
>>> str(pElems[1])
'<p class="slogan">Imparare Python non è mai stato così facile!</p>'
>>> pElems[1].getText()
'Imparare Python non è mai stato così facile!'
>>> str(pElems[2])
'<p>By <span id="author">Al Sweigart</span></p>'
>>> pElems[2].getText()
'By Al Sweigart'
```

Questa volta, `select()` ci dà una lista con tre corrispondenze, che memorizziamo in `pElems`. Utilizzando `str()` su `pElems[0]`, `pElems[1]` e `pElems[2]`, vediamo ciascun elemento come una stringa; usando `getText()` su ciascun elemento possiamo vederne il testo.

## Ottenere dati dagli attributi di un elemento

Il metodo `get()` per oggetti Tag rende semplice accedere ai valori degli attributi di un elemento. Al metodo viene passata una stringa con il nome di un attributo; si ottiene di ritorno il valore di quell'attributo. Utilizzando ancora `example.html`, inserite quanto segue nella shell interattiva:

```
>>> import bs4
>>> soup = bs4.BeautifulSoup(open('example.html'))
```

```
>>> spanElem = soup.select('span')[0]
>>> str(spanElem)
'<span id="author">Al Sweigart</span>'
>>> spanElem.get('id')
'author'
>>> spanElem.get('un_indirizzo_che_non_esiste') == None
True
>>> spanElem.attrs
{'id': 'author'}
```

Qui usiamo `select()` per trovare ogni elemento `<span>` e poi memorizziamo in `spanElem` il primo elemento identificato. Passando a `get()` il nome di attributo `'id'` abbiamo di ritorno il valore dell'attributo, `'author'`.

## Progetto: Ricerca in Google del tipo “mi sento fortunato”

Ogni volta che cerco un argomento in Google, non mi limito a esaminare un solo risultato di ricerca alla volta. Facendo un clic con il pulsante centrale del mouse (o facendo clic mentre tengo premuto Ctrl), apro un certo numero dei primi risultati in una serie di nuove schede del browser, da leggere in seguito.

Effettuo ricerche in Google abbastanza spesso perché questo flusso di lavoro (aprire il browser, cercare un argomento, fare un clic su più risultati) risulti abbastanza noioso. Sarebbe bello se potessi semplicemente scrivere sulla riga di comando un termine di ricerca e fare in modo che il mio computer apra automaticamente un browser con tutti i primi risultati in nuove schede.

Proviamo a scrivere un programma che faccia proprio questo:

- prendere parole chiave dagli argomenti della riga di comando;
- recuperare la pagina dei risultati di ricerca;
- aprire una scheda nel browser per ciascun risultato.

Questo significa che il codice dovrà fare queste cose:

- leggere gli argomenti della riga di comando da `sys.argv`;
- caricare la pagina dei risultati di ricerca con il modulo `requests`;
- trovare i link per ciascun risultato di ricerca;
- chiamare la funzione `webbrowser.open()` per aprire il browser.

Aprite una nuova finestra di file editor e salvate il file con il nome `lucky.py`.

## Passo 1: prendere gli argomenti dalla riga di comando e richiedere la pagina di ricerca

Prima di scrivere del codice, dovete conoscere l'URL della pagina dei risultati di ricerca. Esaminando la barra degli indirizzi del browser dopo aver effettuato una ricerca in Google, potete vedere che la pagina dei risultati ha un URL del tipo [http://www.google.com/search?q=QUI\\_IL\\_TERMINE\\_DI\\_RICERCA](http://www.google.com/search?q=QUI_IL_TERMINE_DI_RICERCA). Il modulo `request` può scaricare questa pagina, poi potete usare `BeautifulSoup` per trovare i link dei diversi risultati di ricerca nel codice HTML.

Infine, userete il modulo `webbrowser` per aprire quei link in altrettante schede del browser. Iniziate il codice in questo modo:

```
#! python3
# lucky.py - Apre vari risultati di ricerca forniti da Google.

import requests, sys, webbrowser, bs4

print('Sto cercando con Google...') # visualizza del testo mentre scarica la pagina Google
res = requests.get('http://google.com/search?q=' + ' '.join(sys.argv[1:]))
res.raise_for_status()
```

# DA FARE: Recuperare i principali link dei risultati di ricerca.

# DA FARE: Aprire una scheda del browser per ciascun risultato.

L'utente specificherà i termini di ricerca utilizzando argomenti da riga di comando nel momento in cui lancerà il programma. Questi argomenti saranno memorizzati come stringhe in una lista in sys.argv.

## Passo 2: trovare tutti i risultati

Ora dovete usare Beautiful Soup per estrarre i link dei principali risultati di ricerca dall'HTML scaricato.

Quale sarà il selettori giusto per questo scopo? Per esempio, non potete cercare semplicemente tutti i tag <a>, perché ci sono molti link che non sono di interesse; dovete invece analizzare la pagina dei risultati di ricerca con gli strumenti per gli sviluppatori messi a disposizione dal browser per cercare un selettori che estragga solo i collegamenti che vi servono.

Dopo aver fatto una ricerca in Google per *Beautiful Soup*, potete aprire gli strumenti per gli sviluppatori del browser e ispezionare alcuni degli elementi della pagina che sono link. Hanno un aspetto incredibilmente complicato, qualcosa di questo genere:

```
<a href="/url?sa=t&amp;rct=j&amp;q=&amp;esrc=s&amp;source=web&amp;cd=1&amp;cad=rja&amp;uact=8&amp;ved=0CCgQFjAA&amp;url=ht
TuLQ&amp;sig2=sdZu6WVIBIVSDrhwrtworMA" onmousedown="return rwt(this,'','1','AFQjCNHAxwplurFOBqg5cehWQEVKi-
TuLQ','sdZu6WVIBIVSDrhwrtworMA','0CCgQFjAA','','event)" data-href="http://www.crummy.com/software/BeautifulSoup/">
<em>BeautifulSoup</em>: We called him Tortoise because he taught us.</a>.
```

Non ha alcuna importanza che l'elemento abbia un aspetto incredibilmente complicato: dovete solo trovare lo schema che seguono tutti i link dei risultati di ricerca. Però questo elemento <a> non sembra avere nulla che lo distingua facilmente da altri elementi <a> che si trovano nella pagina e non sono risultati di ricerca.

Aggiungete al vostro codice quanto segue:

```
#!/usr/bin/python3
# lucky.py - Apre vari risultati di ricerca forniti da Google.
```

```
import requests, sys, webbrowser, bs4
```

```
--righe omesse--
```

```
# Recupera i link dei principali risultati di ricerca.
soup = bs4.BeautifulSoup(res.text)
```

```
# Apre una scheda del browser per ciascun risultato.
linkElems = soup.select('.r a')
```

Se spostate un po' lo sguardo dall'elemento <a>, però, vedrete che c'è un elemento fatto così: <h3>

class="r">.

Se guardate al resto del sorgente HTML, sembra proprio che la classe `r` sia usata solo per i link dei risultati di ricerca. Non c'è bisogno che sappiate che cos'è la classe `r` dei CSS o che cosa fa: la userete semplicemente come un marcatore per individuare l'elemento `<a>` che vi interessa. Potete creare un oggetto BeautifulSoup dal testo HTML della pagina scaricata e poi usare il selettore '`.r a`' per trovare tutti gli elementi `<a>` che si trovano all'interno di un elemento che ha la classe CSS `r`.

## Passo 3: aprire una scheda del browser per ciascun risultato

Infine, diremo al programma di aprire schede del browser per i nostri risultati. Aggiungete quanto segue in fondo al programma:

```
#! python3
# lucky.py - Apre vari risultati di ricerca forniti da Google.

import requests, sys, webbrowser, bs4

--righe omesse--

# Apre una scheda del browser per ciascun risultato.
linkElems = soup.select('.r a')
numOpen = min(5, len(linkElems))
for i in range(numOpen):
    webbrowser.open('http://google.com' + linkElems[i].get('href'))
```

Per default, aprite i primi cinque risultati di ricerca in nuove schede utilizzando il modulo `webbrowser`. L'utente, però, potrebbe anche aver cercato qualcosa che ha dato meno di cinque risultati. La chiamata a `soup.select()` restituisce una lista con tutti gli elementi che corrispondono al selettore '`.r a`', perciò il numero delle schede da aprire sarà 5 o uguale alla lunghezza della lista (a seconda di quale dei due numeri è minore).

La funzione predefinita di Python `min()` restituisce il più piccolo fra due argomenti interi o in virgola mobile che le vengono passati. (Esiste anche una funzione predefinita `max()` che restituisce il maggiore fra due argomenti che le vengono passati.) Potete usare `min()` per stabilire se nella lista si trovano meno di cinque link e memorizzare il numero dei link da aprire in una variabile `numOpen`. Poi potete eseguire un ciclo `for` chimando `range(numOpen)`. A ciascuna iterazione del ciclo, usate `webbrowser.open()` per aprire una nuova scheda nel browser. Notate che il valore dell'attributo `href` negli elementi `<a>` restituiti non ha la parte iniziale `http://google.com`, perciò dovete concatenarla al valore stringa dell'attributo `href`.

Ora potete aprire istantaneamente i primi cinque risultati di Google per, poniamo, *Python programming tutorials* eseguendo `lucky python programming tutorials` dalla riga di comando. (Vedete l'[Appendice B](#) per le informazioni su come eseguire i programmi sotto il vostro sistema operativo.)

## Idee per programmi simili

Il vantaggio dei browser a schede è che si possono aprire facilmente dei link in nuove schede da esaminare in seguito. Un programma che apre automaticamente più link contemporaneamente può essere una buona scorciatoia per varie attività:

- aprire tutte le pagine di prodotto dopo una ricerca in un sito di vendita come Amazon;

- aprire tutti i link delle recensioni di un unico prodotto;
- aprire i link dei risultati per vedere le fotografie dopo una ricerca in un sito di fotografie come Flickr o Imgur.

## Progetto: scaricare tutti i fumetti XKCD

I blog e altri siti web che vengono aggiornati spesso di solito hanno una pagina iniziale in cui sono riportati i post più recenti e magari anche un pulsante *Previous* o *Precedenti* che porta ai post meno recenti. Anche quei post avranno un pulsante simile, e via di questo passo: così potete seguire una traccia che va dalla pagina più recente al primo post pubblicato sul sito. Se volete una copia dei contenuti del sito per poterli leggere quando non siete online, potreste navigare manualmente raggiungendo le singole pagine e salvarle una per una. Sarebbe però un lavoro molto noioso, perciò proviamo a scrivere un programma che svolga questa attività.

XKCD è un fumetto per geek molto seguito, con un sito web che segue proprio questa struttura ([Figura 11.6](http://xkcd.com/)). La pagina iniziale all'indirizzo <http://xkcd.com/> ha un pulsante *Prev* che guida l'utente a ritroso verso i fumetti precedenti. Scaricare ciascun fumetto a mano richiederebbe un'eternità, ma potete creare uno script che lo farà in un paio di minuti. Ecco che cosa deve fare il programma:

- caricare la home page di XKCD;
- salvare l'immagine del fumetto su quella pagina;
- seguire il link *Prev*;
- ripetere finché non raggiunge il primo fumetto.



**Figura 11.6** - La home page del sito XKCD.

Questo significa che il vostro codice dovrà:

- scaricare le pagine con il modulo `requests`;
- trovare l’URL dell’immagine del fumetto per una pagina con `Beautiful Soup`;
- scaricare e salvare l’immagine del fumetto sul disco rigido con `iter_content()`;
- trovare l’URL del link *Prev* e ripetere la procedura.

Aprite una nuova finestra di file editor e salvate il file come `downloadXkcd.py`.

## Passo 1: progettare il programma

Se aprirete nel vostro browser gli strumenti per gli sviluppatori e ispezionate gli elementi sulla pagina, troverete che:

- l’URL del file dell’immagine del fumetto è data dall’attributo `href` di un elemento `<img>`;
- l’elemento `<img>` si trova all’interno di un elemento `<div id="comic">`
- il pulsante *Prev* ha un attributo `HTML rel` con valore `prev`;
- il pulsante *Prev* del primo fumetto è un link all’URL <http://xkcd.com/#>, il che indica che non esistono altre pagine precedenti.

Iniziate così la scrittura del codice:

```
#! python3
# downloadXkcd.py - Scarica tutti i fumetti da XKCD.

import requests, os, bs4

url = 'http://xkcd.com' # url iniziale
os.makedirs('xkcd', exist_ok=True) # salva i fumetti in ./xkcd

while not url.endswith('#'):
    # DA FARE: Scaricare la pagina.

    # DA FARE: Trovare l'url dell'immagine del fumetto

    # DA FARE: Scaricare l'immagine.

    # DA FARE: Salvare l'immagine in ./xkcd.

    # DA FARE: Trovare l'url del pulsante prev.

print('Fatto.')
```

Avrete una variabile `url` che inizia con il valore '<http://xkcd.com>' e viene continuamente aggiornata (in un ciclo `for`) con l'URL del link *Prev* della pagina corrente. A ogni passo nel ciclo, viene scaricato il fumetto che si trova a `url`. Il ciclo finirà quando la variabile `url` conterrà una stringa che termina con '#'. I file delle immagini verranno scaricati in una cartella `xkcd` all'interno della directory di lavoro corrente.

La chiamata `os.makedirs()` garantisce che questa cartella esista e l'argomento per parola chiave `exist_ok=True` impedisce che la funzione lanci un'eccezione se la cartella esiste già. Il resto del codice per ora è costituito solo da commenti che delineano il resto del programma.

## Passo 2: scaricare la pagina web

Implementiamo ora il codice per scaricare la pagina. Aggiungete al programma:

```
#! python3
# downloadXkcd.py - Scarica tutti i fumetti da XKCD.

import requests, os, bs4

url = 'http://xkcd.com' # url iniziale
os.makedirs('xkcd', exist_ok=True) # salva i fumetti in ./xkcd

while not url.endswith('#'):
    # Scarica la pagina.
    print('Sto scaricando la pagina %s...' % url)
    res = requests.get(url)
    res.raise_for_status()

    soup = bs4.BeautifulSoup(res.text)

    # DA FARE: Trovare l'url dell'immagine del fumetto

    # DA FARE: Scaricare l'immagine.

    # DA FARE: Salvare l'immagine in ./xkcd.

    # DA FARE: Trovare l'url del pulsante prev.

print('Fatto.')
```

In primo luogo, viene stampato il valore di `url`, in modo che l'utente sappia quale URL il programma sta per scaricare; poi si usa la funzione `request.get()` del modulo `requests` per scaricarlo. Come sempre, si chiama subito il metodo `raise_for_status()` dell'oggetto `Response` per lanciare un'eccezione e chiudere il programma se qualcosa è andato storto nello scaricamento. Altrimenti, si crea un oggetto `BeautifulSoup` dal testo della pagina scaricata.

## Passo 3: trovare e scaricare l'immagine del fumetto

Aggiungete quanto segue al vostro codice:

```

#! python3
# downloadXkcd.py - Scarica tutti i fumetti da XKCD.

import requests, os, bs4

--righe omesse--

# Trova l'URL dell'immagine del fumetto.
comicElem = soup.select('#comic img')
if comicElem == []:
    print('Non ho trovato alcuna immagine.')
else:
    comicUrl = 'http:' + comicElem[0].get('src')
    # Scarica l'immagine.
    print('Sto scaricando l\'immagine %s...' % (comicUrl))
    res = requests.get(comicUrl)
    res.raise_for_status()

# DA FARE: Salvare l'immagine in ./xkcd.

# DA FARE: Trovare l'url del pulsante prev.

print('Fatto.')

```

Avendo ispezionato la home page di XKCD con gli strumenti per gli sviluppatori, sapete che l'elemento `<img>` dell'immagine del fumetto si trova all'interno di un elemento `<div>` con attributo id impostato a `comic`, perciò il selettore `'#comic img'` darà l'elemento `<img>` corretto dall'oggetto BeautifulSoup. Qualche pagina XKCD ha contenuti speciali che non sono un semplice file di immagine. Va bene: li salterete.

Se il selettore non trova elementi, `soup.select('#comic img')` restituirà una lista vuota. Il programma allora può semplicemente stampare un messaggio d'errore e passare oltre senza scaricare l'immagine.

Altrimenti, il selettore restituirà una lista con un elemento `<img>`. Potete prendere l'attributo `src` da questo elemento `<img>` e passarlo a `requests.get()` per scaricare il file dell'immagine del fumetto.

## Passo 4: salvare l'immagine e trovare il fumetto precedente

Integrate il vostro codice in questo modo:

```
#! python3
# downloadXkcd.py - Scarica tutti i fumetti da XKCD.

import requests, os, bs4

--righe omesse--

    # Salva l'immagine in ./xkcd.
    imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)), 'wb')
    for chunk in res.iter_content(100000):
        imageFile.write(chunk)
    imageFile.close()

    # Trova l'url del pulsante prev.
    prevLink = soup.select('a[rel="prev"]')[0]
    url = 'http://xkcd.com' + prevLink.get('href')

print('Fatto.')
```

A questo punto, il file dell'immagine del fumetto è memorizzata nella variabile `res`. Dovete scrivere questi dati d'immagine in un file sul disco fisso.

Servirà un nome per il file dell'immagine locale da passare a `open()`. La variabile `comicUrl` avrà un valore come '[http://imgs.xkcd.com/comics/all\\_you\\_can\\_eat.png](http://imgs.xkcd.com/comics/all_you_can_eat.png)' – che, come avrete notato, è molto simile a un percorso di file. In effetti, potete chiamare `os.path.basename()` con `comicUrl` e avrete di ritorno proprio l'ultima parte dell'URL, 'all\_you\_can\_eat.png', che potete usare come nome di file quando salvate l'immagine sul disco fisso. Accodate questo nome al nome della cartella `xkcd`, utilizzando `os.path.join()` in modo che il vostro programma usi le barre retroverse (\) in Windows e quelle normali (/) in OS X e Linux. Ora che finalmente avete il nome del file, potete chiamare `open()` per aprire un nuovo file in modalità 'wb', scrittura binaria.

Ricorderete, da quanto abbiamo detto in precedenza in questo capitolo, che per salvare file scaricati mediante Requests bisogna ciclare sul valore restituito dal metodo `iter_content()`. Il codice nel ciclo `for` scrive parti dei dati d'immagine (al massimo 100.000 byte ogni volta) sul file e poi il file verrà chiuso. L'immagine ora è salvata sul disco fisso.

Poi, il selettori 'a[rel="prev"]' identifica l'elemento `<a>` con l'attributo `rel` impostato a `prev`, e si può usare l'attributo `href` di questo elemento `<a>` per ottenere l'URL del fumetto precedente, che viene memorizzato in `url`. Poi il ciclo `while` inizia di nuovo tutto il processo di scaricamento di questo fumetto.

L'output del programma sarà come questo:

```
Sto scaricando la pagina http://xkcd.com...
Sto scaricando l'immagine http://imgs.xkcd.com/comics/all\_you\_can\_eat.png...
Sto scaricando la pagina http://xkcd.com/1794/...
Sto scaricando l'immagine http://imgs.xkcd.com/comics/fire.png...
Sto scaricando la pagina http://xkcd.com/1793/...
Sto scaricando l'immagine http://imgs.xkcd.com/comics/soda\_sugar\_comparisons.png...
Sto scaricando la pagina http://xkcd.com/1792/...
Sto scaricando l'immagine http://imgs.xkcd.com/comics/bird\_plane\_superman.png...
Sto scaricando la pagina http://xkcd.com/1791/...
Sto scaricando l'immagine http://imgs.xkcd.com/comics/telescopes\_refractor\_vs\_reflector.png...
Sto scaricando la pagina http://xkcd.com/1790/...
Sto scaricando l'immagine http://imgs.xkcd.com/comics/sad.png...
-- e via di questo passo--
```

Fatto.

(Nel momento in cui scriviamo, i file di immagine in XKCD sono 1794 e quando leggerete questa pagina saranno sicuramente di più. Se non volete scaricarli tutti, visto che comunque il computer impiega un po' di tempo e che ogni file occupa un po' di spazio, potete sostituire in while not url.endswith('#') la stringa '#' con una stringa contenente il numero dell'ultima immagine che volete scaricare, per esempio '1700'. Così tornerete indietro solo fino a un certo punto.)

Questo progetto è un buon esempio di un programma che può seguire automaticamente dei link per recuperare grandi quantità di dati dal Web. Potete scoprire le altre caratteristiche di Beautiful Soup consultando la sua documentazione all'indirizzo <http://www.crummy.com/software/BeautifulSoup/bs4/doc/>.

## Idee per programmi simili

Scaricare pagine e seguire i link sono le operazioni fondamentali di molti programmi di web crawling. Programmi simili potrebbero anche:

- effettuare il backup di un intero sito seguendo tutti i suoi link;
- copiare tutti i messaggi di un forum web;
- duplicare il catalogo dei prodotti in vendita in un negozio online.

I moduli requests e BeautifulSoup funzionano benissimo, se potete stabilire quale sia l'URL da passare a requests.get(). A volte però identificarlo non è tanto facile, o magari il sito web che volete far esplorare al vostro programma richiede prima un login. In tal caso, il modulo selenium darà ai vostri programmi gli strumenti necessari anche per svolgere attività così raffinate.

## Controllare il browser con il modulo selenium

Il modulo selenium permette a Python di controllare direttamente il browser, facendo sì che da programma sia possibile fare clic sui link e compilare le informazioni di login, come se ci fosse un utente umano che interagisce con la pagina. Il modulo selenium permette di interagire con le pagine web in modo molto più complesso di quanto non consentano Requests e Beautiful Soup; poiché però lancia un browser è un po' più lento ed è difficile farlo girare in background se, poniamo, si devono solamente scaricare dei file dal Web. (L'[Appendice A](#) presenta informazioni dettagliate su come installare moduli di terze parti.)

## Avvio di un browser controllato da selenium

Per questi esempi, vi servirà il browser Firefox: sarà questo il browser che controllerete. Se non disponete di Firefox, potete scaricarlo gratuitamente dall'indirizzo <http://getfirefox.com/>.

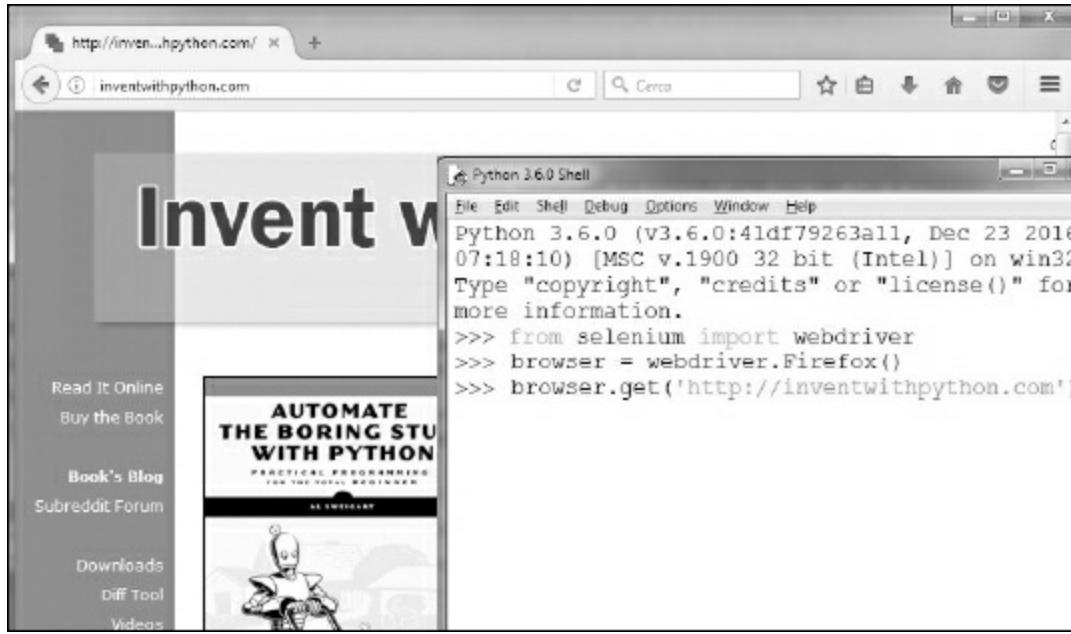
L'importazione dei moduli per Selenium è un po' tortuosa. Anziché import selenium, dovete eseguire from selenium import webdriver. (Il motivo preciso per cui il modulo selenium è impostato in questo modo va al di là del raggio d'azione di questo libro.) Dopo questo, potete lanciare il browser Firefox con Selenium.

Inserite quanto segue nella shell interattiva:

```
>>> from selenium import webdriver
```

```
>>> browser = webdriver.Firefox()
>>> type(browser)
<class 'selenium.webdriver.firefox.webdriver.WebDriver'>
>>> browser.get('http://inventwithpython.com')
```

Noterete che, quando viene chiamata `webdriver.Firefox()`, si avvia il browser Firefox. La chiamata a `type()` sul valore `webdriver.Firefox()` ci dice che è del tipo di dati WebDriver. Chiamando `browser.get('http://inventwithpython.com')` si punta il browser all'indirizzo <http://inventwithpython.com/>. La finestra del browser sarà come nella [Figura 11.7](#).



**Figura 11.7** - Dopo aver chiamato `webdriver.Firefox()` e `get()` in IDLE, si apre il browser Firefox.

## Trovare elementi sulla pagina

Gli oggetti WebDriver possiedono vari metodi per trovare elementi in una pagina. Si suddividono in metodi `find_element_*` e `find_elements_*`: i primi restituiscono un singolo oggetto WebElement, che rappresenta il primo elemento sulla pagina che soddisfa l'interrogazione. I metodi `find_elements_*` invece restituiscono una lista di oggetti WebElement per tutti gli elementi che soddisfano l'interrogazione presenti nella pagina.

La [Tabella 11.3](#) mostra vari esempi di metodi `find_element_*` e `find_elements_*` chiamati su un oggetto WebDriver memorizzato nella variabile `browser`.

**Tabella 11.3** - I metodi di WebDriver di Selenium per trovare elementi.

Nome del metodo	Oggetto WebElement o lista di WebElement restituiti
<code>browser.find_element_by_class_name(nome)</code> <code>browser.find_elements_by_class_name(nome)</code>	Elementi che usano la classe CSS <i>nome</i>
<code>browser.find_element_by_css_selector(selettore)</code> <code>browser.find_elements_by_css_selector(selettore)</code>	Elementi che corrispondono al <i>selettore</i> CSS
<code>browser.find_element_by_id(id)</code> <code>browser.find_elements_by_id(id)</code>	Elementi con un valore dell'attributo id che corrisponde alla ricerca
<code>browser.find_element_by_link_text(testo)</code>	

<code>browser.find_elements_by_link_text(testo)</code>	Elementi <a> che corrispondono completamente al testo fornito
<code>browser.find_element_by_partial_link_text(testo)</code> <code>browser.find_elements_by_partial_link_text(testo)</code>	Elemento <a> che contiene il testo fornito
<code>browser.find_element_by_name(nome)</code> <code>browser.find_elements_by_name(nome)</code>	Elementi con un valore dell'attributo <i>nome</i> corrispondente.
<code>browser.find_element_by_tag_name(nome)</code> <code>browser.find_elements_by_tag_name(nome)</code>	Elementi con <i>nome</i> di tag (non distingue maiuscole e minuscole: un elemento <a> è identificato da 'a' e da 'A')

Fatta eccezione per i metodi `*_by_tag_name()`, gli argomenti di tutti i metodi fanno distinzione fra maiuscole e minuscole. Se nella pagina non esiste alcun elemento che corrisponda a ciò che il metodo sta cercando, il modulo selenium solleva un'eccezione `NoSuchElement`. Se non volete che questa eccezione mandi in crash il vostro programma, aggiungete al vostro codice enunciati `try` ed `except`. Una volta che avete l'oggetto `WebElement`, potete trovare maggiori informazioni su quell'oggetto leggendo gli attributi o chiamando i metodi della [Tabella 11.4](#).

**Tabella 11.4** - Attributi e metodi di `WebElement`.

Attributo o metodo	Descrizione
<code>tag_name</code>	Il nome del tag, per esempio 'a' per un elemento <a>
<code>get_attribute(nome)</code>	Il valore per l'attributo <i>nome</i> dell'elemento
<code>text</code>	Il testo nell'elemento, per esempio 'ciao' in <span>ciao</span>
<code>clear()</code>	Per elementi campo testo o area testo, elimina il testo che vi è stato scritto
<code>is_displayed()</code>	Restituisce True se l'elemento è visibile; altrimenti False
<code>is_enabled()</code>	Per elementi di input, restituisce True se l'elemento è attivato; altrimenti False
<code>is_selected()</code>	Per elementi casella di spunta o casella di opzione, restituisce True se l'elemento è selezionato; altrimenti False
<code>location</code>	Un dizionario con le chiavi 'x' e 'y' per la posizione dell'elemento nella pagina

Per esempio, aprite una nuova finestra di file editor e inserite questo programma:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://inventwithpython.com')

try:
    elem = browser.find_element_by_class_name('bookcover')
    print('Trovato elemento <%s> con quel nome di classe!' % (elem.tag_name))
except:
    print('Non sono riuscito a trovare elementi con quel nome.')
```

Qui apriamo Firefox e lo puntiamo a un URL. Su questa pagina, cerchiamo elementi con il nome di classe 'bookcover' e, se un elemento del genere viene trovato, ne stampiamo il nome di tag utilizzando l'attributo `tag_name`.

Se non viene trovato alcun elemento, invece, stampiamo un messaggio diverso. L'output del programma sarà di questo tipo:

Trovato elemento <img> con quel nome di classe!

Abbiamo trovato un elemento con nome di classe 'bookcover' e nome di tag 'img'.

## Fare clic sulla pagina

Gli oggetti WebElement restituiti dai metodi `find_element_*` e `find_elements_*` hanno un metodo `click()` che simula un clic del mouse su quell'elemento. Questo metodo può essere usato per seguire un link, effettuare la selezione di un pulsante di opzione, fare clic su un pulsante Invia, o dare il via a qualsiasi attività sia associata al clic del mouse su quell'elemento. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> from selenium import webdriver  
>>> browser = webdriver.Firefox()  
>>> browser.get('http://inventwithpython.com')  
>>> linkElem = browser.find_element_by_link_text('Read It Online')  
>>> type(linkElem)  
<class 'selenium.webdriver.remote.webelement.WebElement'>  
>>> linkElem.click()      # segue il link "Read It Online" link
```

Questo apre Firefox sulla pagina <http://inventwithpython.com/>, prende l'oggetto WebElement per l'elemento <a> con il testo *Read It Online*, poi simula un clic sull'elemento <a>. È come se aveste fatto clic sul link voi stessi; il browser segue quel link.

## Compilare e inoltrare moduli

Per poter inviare pressioni di tasto a campi di testo su una pagina web bisogna trovare l'elemento <input> o <textarea> per quel campo di testo e poi chiamare il metodo `send_keys()`.

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> from selenium import webdriver  
>>> browser = webdriver.Firefox()  
>>> browser.get('https://mail.yahoo.com')  
>>> emailElem = browser.find_element_by_id('login-username')  
>>> emailElem.send_keys('not_my_real_email')  
>>> passwordElem = browser.find_element_by_id('login-passwd')  
>>> passwordElem.send_keys('12345')  
>>> passwordElem.submit()
```

A meno che Gmail non abbia modificato l'id dei campi *Nome utente* e *Password* dopo che questo libro è stato pubblicato, questo codice compilerà quei campi con il testo indicato. (Potete sempre usare la funzione di ispezione del browser per verificare l'id.) Chiamando il metodo `submit()` su qualsiasi elemento si otterrà lo stesso risultato di un clic sul pulsante *Invia* (o *Submit*) per il modulo in cui si trova quell'elemento. (Avreste potuto chiamare altrettanto facilmente `emailElem.submit()` e il codice avrebbe fatto la stessa cosa.)

## Invio di tasti speciali

Selenium ha un modulo per tasti della tastiera che è impossibile scrivere in un valore stringa, che funziona un po' come i caratteri escape. Questi valori sono memorizzati in attributi nel modulo selenium.webdriver.common.keys. Dato che il nome del modulo è così lungo, è molto più facile eseguire from selenium.webdriver.common.keys import Keys all'inizio del programma; in questo caso, poi potrete semplicemente scrivere Keys dovunque avreste dovuto normalmente scrivere selenium.webdriver.common.keys. La [Tabella 11.5](#) elenca le variabili Keys di uso più comune.

**Tabella 11.5** - Variabili di uso comune nel modulo selenium.webdriver.common.keys.

Attributi	Significato
Keys.DOWN, Keys.UP, Keys.LEFT, Keys.RIGHT	I tasti con le frecce
Keys.ENTER, Keys.RETURN	I tasti Invio da tastiera e tastierino
Keys.HOME, Keys.END, Keys.PAGE_DOWN, Keys.PAGE_UP	I tasti Home, Fine, PGSU e PGGLù
Keys.ESCAPE, Keys.BACK_SPACE, Keys.DELETE	I tasti Esc, Backspace e Canc
Keys.ESCAPE, Keys.BACK_SPACE, Keys.DELETE	I tasti da F1 a F12
Keys.TAB	Il tasto Tab

Per esempio, se il cursore al momento non si trova in un campo di testo, la pressione dei tasti Home e Fine farà scorrere il browser in testa o in fondo alla pagina, rispettivamente. Inserite quanto segue nella shell interattiva e notate come le chiamate `send_keys` effettuino lo scorrimento della pagina:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.keys import Keys
>>> browser = webdriver.Firefox()
>>> browser.get('http://nostarch.com')
>>> htmlElem = browser.find_element_by_tag_name('html')
>>> htmlElem.send_keys(Keys.END) # scorre in fondo
>>> htmlElem.send_keys(Keys.HOME) # scorre in cima
```

Il tag `<html>` è il tag fondamentale nei file HTML: tutto il contenuto di un documento HTML deve essere racchiuso fra i tag `<html>` e `</html>`. La chiamata `browser.find_element_by_tag_name('html')` è un buon posto per inviare tasti alla pagina web generale. questo procedimento sarebbe utile se, per esempio, venissero caricati nuovi contenuti dopo che avete effettuato lo scorrimento fino al fondo della pagina.

## Fare clic sui pulsanti del browser

Selenium può simulare anche i clic sui vari pulsanti del browser, mediante i metodi seguenti:

- `browser.back()` clic sul pulsante *Indietro*.
- `browser.forward()` clic sul pulsante *Avanti*.
- `browser.refresh()` clic sul pulsante *Ricarica*.
- `browser.quit()` clic sul pulsante di chiusura della finestra.

## Maggiori informazioni su Selenium

Selenium può fare molte altre cose, oltre le funzioni descritte qui. Può modificare i cookie del browser, catturare schermate delle pagine web ed eseguire script personalizzati di JavaScript. Per scoprire di più su queste caratteristiche, potete consultare la documentazione di Selenium all'indirizzo <http://selenium-python.readthedocs.org/>.

## Riepilogo

Le attività noiose non sono solo quelle che riguardano i file presenti sul computer. Sapere **scaricare pagine web** da programma estenderà le vostre capacità di programmazione a Internet. Il modulo requests rende immediato il download e, con qualche conoscenza di base dei concetti e dei selettori dell'HTML, potete utilizzare il modulo BeautifulSoup per analizzare le pagine che scaricate.

Per automatizzare completamente ogni attività basata sul Web, vi serve però il controllo diretto del browser web attraverso il modulo selenium. Questo modulo vi consentirà di effettuare il login ai siti web e di compilare moduli automaticamente. Un browser è lo strumento più comune per inviare e ricevere informazioni via Internet, perciò questa è una competenza importante nella cassetta dei vostri strumenti da programmatore.

## Domande di ripasso

1. Descrivete brevemente le differenze fra i moduli webbrowser, requests, BeautifulSoup e selenium.
2. Che tipo di oggetto viene restituito da `requests.get()`? Come potete accedere ai contenuti scaricati sotto forma di valore stringa?
3. Quale metodo di `requests` verifica che il download sia andato a buon fine?
4. Come potete ottenere il codice di status HTTP di una risposta a `requests`?
5. Come si salva una risposta di `requests` in un file?
6. Qual è la sorciatoia da tastiera per aprire gli strumenti per gli sviluppatori in un browser web?
7. Come si può vedere (negli strumenti per gli sviluppatori) l'HTML di uno specifico elemento in una pagina web?
8. Qual è la stringa selettore CSS che troverebbe l'elemento con un attributo `id` uguale a `main`?
9. Qual è la stringa selettore CSS che troverebbe gli elementi con una classe CSS `highlight`?
10. Qual è la stringa selettore CSS che troverebbe tutti gli elementi `<div>` all'interno di un altro elemento `<div>`?
11. Qual è la stringa selettore CSS che troverebbe l'elemento `<button>` con un attributo `value` impostato a `favorite`?
12. Supponiamo che abbiate un oggetto Tag di BeautifulSoup memorizzato nella variabile `spam` per l'elemento `<div>Ciao mondo!</div>`. Come potreste ricavare una stringa 'Ciao mondo!' dall'oggetto Tag?
13. Come memorizzereste tutti gli attributi di un oggetto Tag di BeautifulSoup in una variabile `linkElem`?
14. L'esecuzione di `import selenium` non funziona. Qual è il modo corretto per importare il modulo selenium?
15. Qual è la differenza fra i metodi `find_element_*` e `find_elements_*`?
16. Quali metodi hanno gli oggetti WebElement di Selenium per simulare i clic del mouse e la pressione di tasti sulla tastiera?
17. Potreste chiamare `send_keys(Keys.ENTER)` sull'oggetto WebElement del pulsante *Invia* (o *Submit*)?

ma qual è un modo più facile per inviare con Selenium un modulo compilato online?

18. Come potete simulare con Selenium il clic sui pulsanti *Avanti*, *Indietro* e *Ricarica* di un browser?

## Un po' di pratica

Per esercitarvi, scrivete dei programmi che svolgano le attività seguenti.

### Email da riga di comando

Scrivete un programma che sulla riga di comando prenda un indirizzo email e una stringa di testo e poi, utilizzando Selenium, effettui il login al vostro account di posta elettronica e invii un messaggio costituito da quella stringa all'indirizzo fornito. (Magari potete creare un account di posta appositamente per questo programma.)

Sarebbe un bel modo per aggiungere una funzione di notifica ai vostri programmi. Potreste scrivere un programma analogo anche per inviare messaggi da un account di Facebook o di Twitter.

### Scaricamento da siti di immagini

Scrivete un programma che visiti un sito di condivisione di foto come Flickr o Imgur, cerchi una categoria di fotografia e poi scarichi tutte le immagini risultanti. Potreste scrivere un programma che funzioni con qualsiasi sito di fotografie che disponga di una funzione di ricerca.

### 2048

2048 è un gioco semplice in cui si combinano piastrelline facendole scorrere verso l'alto, il basso, a sinistra o a destra con i tasti freccia. Si può raggiungere un punteggio abbastanza elevato facendo scorrere continuamente secondo uno schema su, a destra, giù e a sinistra. Scrivete un programma che apra il gioco all'indirizzo <https://gabrielecirulli.github.io/2048/> e continui a inviare i tasti su, destra, giù e sinistra per giocare automaticamente.

### Verifica di link

Scrivete un programma che, dato un URL, cerchi di scaricare tutte le pagine a cui puntano i link presenti su quella pagina web. Il programma deve evidenziare le pagine con un codice di status *404 “Not Found”* e stamparne l'indirizzo in quanto link interrotto.

---

<sup>1</sup> La risposta è no.

# Lavorare con fogli di calcolo Excel

**Excel** è un'applicazione per Windows, molto diffusa e potente, per la creazione di **fogli di calcolo**. Il modulo **openpyxl** permette ai programmi Python di **leggere e modificare file** di Excel: per esempio, si possono **copiare** certi dati da un foglio di calcolo, per poi incollarli in un altro; oppure si possono **selezionare** poche righe fra le migliaia di un foglio di lavoro, in base a qualche criterio, per apportarvi qualche modifica; fra centinaia di fogli che rappresentano budget, si possono **trovare** quelli con i conti in rosso.

Sono solo alcuni dei compiti noiosi e intellettualmente poco stimolanti che hanno a che fare con i fogli di calcolo e che Python può aiutarvi ad automatizzare.

Excel è software proprietario della Microsoft, ma esistono altre alternative libere che girano sotto Windows, OS X e Linux. Sia LibreOffice Calc, sia OpenOffice Calc lavorano con il formato di file .xlsx di Excel per i fogli di calcolo, il che significa che il modulo openpyxl può lavorare anche con i fogli di calcolo creati con queste applicazioni. Potete scaricare questi software da <https://www.libreoffice.org> e <http://www.openoffice.org>, rispettivamente. Anche se avete già installato Excel sul vostro computer, potreste trovare questi programmi più facili da usare. Le schermate che vedrete in questo capitolo, comunque, sono state ricavate con Excel 2010 sotto Windows 7.

## Documenti Excel

Innanzitutto, qualche definizione di base: un documento creato con Excel prende il nome di **cartella di lavoro**. Una singola cartella di lavoro viene salvata in un file con estensione .xlsx. Ciascuna

cartella può contenere più **fogli** (detti anche **fogli di lavoro**). Il foglio che l’utente visualizza in un dato momento (o l’ultimo che ha visualizzato prima di chiudere Excel) è chiamato **foglio attivo**. Ciascun foglio è organizzato in **colonne** (identificate da lettere, a partire da *A*) e **righe** (identificate da numeri, a partire da 1). Il riquadro che si trova all’incrocio di una particolare riga e di una particolare colonna è chiamato **cella**. Ogni cella può contenere un numero o un valore di testo. La griglia di celle contenenti dati costituisce un foglio.

## Installazione del modulo openpyxl

Python non è fornito “di serie” con OpenPyXL, perciò è necessario installarlo. Seguite le istruzioni per l’installazione di moduli di terze parti nell’[Appendice A](#); il nome del modulo è `openpyxl`. Fate attenzione: scrivete `pip install opnpyxl=2.1.4` per installare la versione 2.1.4, altrimenti il codice di questo capitolo non funzionerà. Per verificare se è stato installato correttamente, inserite poi quanto segue nella shell interattiva:

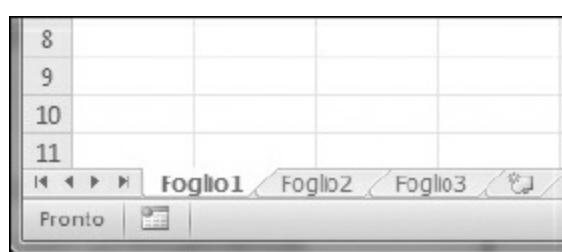
```
>>> import openpyxl
```

Se il modulo è stato installato correttamente, questo non dovrebbe dare alcun messaggio d’errore. Ricordate di importare il modulo `openpyxl` prima di eseguire gli esempi nella shell interattiva in questo capitolo, altrimenti riceverete un `NameError: name 'openpyxl' is not defined`.

Questo libro tratta la versione 2.1.4 di OpenPyXL, ma il team di OpenPyXL rilascia regolarmente nuove versioni; le versioni successive non sono compatibili con la 2.1.4, perciò se avete una versione successiva non tutti gli esempi del capitolo funzioneranno. Se volete vedere quali ulteriori caratteristiche vi metta a disposizione una versione successiva, potete consultare la documentazione completa di OpenPyXL all’indirizzo <http://openpyxl.readthedocs.org/>.

## Leggere documenti Excel

Gli esempi in questo capitolo useranno un foglio di calcolo `example.xlsx`, memorizzato nella cartella radice. Potete o creare direttamente il foglio di calcolo oppure scaricarlo da <http://nostarch.com/automatestuff/>. La [Figura 12.1](#) mostra le schede per i tre fogli di default, chiamati *Foglio1*, *Foglio2* e *Foglio3*, che Excel crea automaticamente per le nuove cartelle di lavoro. (Il numero dei fogli creati di default può variare da un sistema operativo all’altro e da un programma all’altro.)



**Figura 12.1** - Le schede per i fogli di una cartella di lavoro si trovano nell’angolo inferiore sinistro della finestra del programma.

Il Foglio 1 nel file di esempio deve essere come quello riportato nella [Tabella 12.1](#). (Se non avete scaricato `example.xlsx` dal sito web, dovete inserire questi dati direttamente nel vostro foglio di lavoro.)

**Tabella 12.1** - Il foglio di calcolo example.xlsx.

A	B	C
1	4/5/2015 1:34:02 PM	Apples
2	4/5/2015 3:41:23 AM	Cherries
3	4/6/2015 12:46:51 PM	Pears
4	4/8/2015 8:59:43 AM	Oranges
5	4/10/2015 2:07:00 AM	Apples
6	4/10/2015 6:10:37 PM	Bananas
7	4/10/2015 2:40:46 AM	Strawberries

Ora che abbiamo il nostro foglio di esempio, vediamo come possiamo manipolarlo con il modulo openpyxl.

## Aprire documenti Excel con OpenPyXL

Importato il modulo openpyxl, potete usare la funzione `openpyxl.load_workbook()`. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> type(wb)
<class 'openpyxl.workbook.workbook.Workbook'>
```

La funzione `openpyxl.load_workbook()` prende come argomento il nome del file e restituisce un valore del tipo di dati `workbook`. Questo oggetto Wokbook rappresenta il file Excel, un po' come un oggetto `File` rappresenta un file di testo aperto.

Ricordate che `example.xlsx` deve essere nella directory di lavoro corrente, perché possiate lavorarci. Potete scoprire quale sia la directory di lavoro corrente importando `os` e usando `os.getcwd()`, e potete cambiare la directory di lavoro corrente con `os.chdir()`.

## Estrarre fogli dalla cartella di lavoro

Potete ottenere una lista di tutti i nomi dei fogli nella cartella di lavoro chiamando il metodo `get_sheet_names()`. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> wb.get_sheet_names()
['Foglio1', 'Foglio2', 'Foglio3']
>>> sheet = wb.get_sheet_by_name('Foglio3')
>>> sheet
<Worksheet "Foglio3">
>>> type(sheet)
<class 'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title
'Foglio3'
>>> anotherSheet = wb.get_active_sheet()
>>> anotherSheet
```

Ciascun foglio è rappresentato da un oggetto Worksheet, che potete ottenere passando la stringa con il nome del foglio al metodo `get_sheet_by_name()` per la cartella di lavoro. Infine potete chiamare il metodo `get_active_sheet()` di un oggetto Workbook per ottenere il foglio attivo di quella cartella. Il foglio attivo è il foglio che viene visualizzato quando si apre la cartella di lavoro in Excel. Una volta che avete l'oggetto Worksheet, potete ottenere il suo nome dall'attributo `Title`.

## Estrarre le celle dai fogli

Una volta che avete un oggetto Worksheet, potete accedere a un oggetto Cell mediante il suo nome (dato dalla lettera della colonna e dal numero della riga al cui incrocio quella cella si trova). Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Foglio1')
>>> sheet['A1']
<Cell Foglio1.A1>
>>> sheet['A1'].value
datetime.datetime(2015, 4, 5, 13, 34, 2)
>>> c = sheet['B1']
>>> c.value
'Apples'
>>> 'La riga ' + str(c.row) + ', colonna ' + c.column + ' è ' + c.value
'La riga 1, colonna B è Apples'
>>> 'La cella ' + c.coordinate + ' contiene la parola ' + c.value
'La cella B1 contiene la parola Apples'
>>> sheet['C1'].value
73
```

L'oggetto Cell ha un attributo `value` che contiene (non è una sorpresa) il valore memorizzato in quella cella. Gli oggetti Cell hanno anche attributi `row`, `column` e `coordinate` che danno le informazioni sulla posizione di quella cella.

Qui, accedendo all'attributo `value` del nostro oggetto Cell per la cella B1 si ottiene la stringa 'Apples'. L'attributo `row` ci dà l'intero 1, l'attributo `column` ci dà 'B' e l'attributo `coordinate` ci dà 'B1'.

OpenPyXL interpreta automaticamente le date nella colonna A e le restituisce come valori `datetime` invece che come stringhe. Del tipo di dati `datetime` ci occuperemo specificamente nel [Capitolo 16](#).

Specificare una colonna mediante la lettera identificativa può essere complicato, in particolare perché, dopo la Z, le colonne iniziano a usare due lettere: AA, AB, AC e così via. In alternativa, potete accedere a una cella usando il metodo `cell()` del foglio e passando degli interi per i suoi argomenti per parola chiave `row` e `column`. La prima riga e la prima colonna sono rappresentate dall'intero 1, non dallo 0. Continuate l'esempio nella shell interattiva inserendo quanto segue:

```
>>> sheet.cell(row=1, column=2)
<Cell Foglio1.B1>
>>> sheet.cell(row=1, column=2).value
'Apples'
>>> for i in range (1, 8, 2):
    print(i, sheet.cell(row=i, column=2).value)
```

```
1 Apples
3 Pears
5 Apples
7 Strawberries
```

Come potete vedere, utilizzando il metodo `cell()` del foglio e passandogli `row=1` e `column=2` si ottiene un oggetto `Cell` per la cella B1, proprio come quando si è specificato `sheet['B1']`. Poi, utilizzando il metodo `cell()` e i suoi argomenti per parola chiave, si può scrivere un ciclo `for` per stampare i valori di una serie di celle.

Supponiamo che vogliate prendere la colonna B e stampare il valore di ogni cella con un numero di riga dispari. Passando 2 come parametro “passo” per la funzione `range()`, si ottengono tutte le righe di numero dispari. La variabile `i` del ciclo `for` viene passata come argomento per parola chiave `row` al metodo `cell()`, mentre 2 viene sempre passato come argomento per parola chiave `column`. Notate che viene passato l’intero 2, non la stringa ‘B’.

Potete stabilire le dimensioni del foglio con i metodi `get_highest_row()` e `get_highest_column()` dell’oggetto `Worksheet`. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet.get_highest_row()
7
>>> sheet.get_highest_column()
3
```

Notate che il metodo `get_highest_column()` restituisce un intero, anziché la lettera che compare in Excel.

## Conversioni fra lettere e numeri di colonna

Per passare dalle lettere ai numeri nella denominazione delle colonne, chiama la funzione `openpyxl.cell.column_index_from_string()`. Per convertire da numeri a lettere, chiamate la funzione `openpyxl.cell.get_column_letter()`. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl
>>> from openpyxl.cell import get_column_letter, column_index_from_string
>>> get_column_letter(1)
'A'
>>> get_column_letter(2)
'B'
>>> get_column_letter(27)
'AA'
>>> get_column_letter(900)
'AHP'
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Foglio1')
>>> get_column_letter(sheet.get_highest_column())
'C'
>>> column_index_from_string('A')
1
>>> column_index_from_string('AA')
27
```

Dopo aver importato da `openpyxl.cell` queste due funzioni, si può chiamare `get_column_letter()` e passarle un intero come 27 per scoprire quale sia la lettera che identifica la 27-esima colonna. La funzione `column_index_from_string()` fa il contrario: prende la lettera di una colonna e vi dice qual è il numero di colonna corrispondente. Non è necessario avere caricato una cartella di lavoro per usare queste funzioni. Se volete, potete caricare una cartella di lavoro, ottenere un oggetto `Worksheet` e chiamare un metodo come `get_highest_column()` per ottenere un intero, da passare poi a `get_column_letter()`.

## Estrarre righe e colonne dai fogli

Potete “sezionare” gli oggetti `Worksheet` per ottenere tutti gli oggetti `Cell` presenti in una riga, una colonna o un’area rettangolare del foglio. Poi potete avviare un ciclo su tutte le celle di quella sezione.

Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl  
>>> wb = openpyxl.load_workbook('example.xlsx')  
>>> sheet = wb.get_sheet_by_name('Foglio1')  
>>> tuple(sheet['A1':'C3'])  
((<Cell Foglio1.A1>, <Cell Foglio1.B1>, <Cell Foglio1.C1>), (<Cell Foglio1.A2>,  
<Cell Foglio1.B2>, <Cell Foglio1.C2>), (<Cell Foglio1.A3>, <Cell Foglio1.B3>,  
<Cell Foglio1.C3>))  
❶ >>> for rowOfCellObjects in sheet['A1':'C3']:  
❷     for cellObj in rowOfCellObjects:  
         print(cellObj.coordinate, cellObj.value)  
     print('---FINE DELLA RIGA ---')
```

```
A1 2015-04-05 13:34:02  
B1 Apples  
C1 73  
---FINE DELLA RIGA ---  
A2 2015-04-05 03:41:23  
B2 Cherries  
C2 85  
---FINE DELLA RIGA ---  
A3 2015-04-06 12:46:51  
B3 Pears  
C3 14  
---FINE DELLA RIGA ---
```

Qui specifichiamo che vogliamo gli oggetti `Cell` nell’area rettangolare da A1 a C3 e otteniamo un oggetto `Generator` che contiene gli oggetti `Cell` in quell’area. Come ausilio per visualizzare questo oggetto `Generator`, possiamo usare `tuple()` su di esso per visualizzare gli oggetti `Cell` in una tupla.

Questa tupla contiene a sua volta tre tuple: una per ciascuna riga, procedendo dalla più alta verso il basso. Ciascuna di queste tre tuple interne contiene gli oggetti `Cell` presenti in una riga dell’area desiderata, procedendo da sinistra verso destra. Nel complesso, quindi la nostra sezione del foglio di calcolo contiene tutti gli oggetti `Cell` nell’area compresa fra A1 e C3, partendo dalla cella superiore sinistra, per finire con quella inferiore destra.

Per stampare i valori di ciascuna cella in quell’area, usiamo due cicli `for`. Quello esterno percorre ciascuna riga della sezione ❶. Poi, per ciascuna riga, il ciclo `for` annidato percorre ciascuna cella in quella riga ❷.

Per accedere ai valori delle celle in una particolare riga o colonna, potete usare anche gli attributi

rows e columns di un oggetto Worksheet. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl  
>>> wb = openpyxl.load_workbook('example.xlsx')  
>>> sheet = wb.get_active_sheet()  
>>> sheet.columns[1]  
(<Cell Foglio1.B1>, <Cell Foglio1.B2>, <Cell Foglio1.B3>, <Cell Foglio1.B4>,  
<Cell Foglio1.B5>, <Cell Foglio1.B6>, <Cell Foglio1.B7>)  
>>> for cellObj in sheet.columns[1]:  
    print(cellObj.value)
```

Apples  
Cherries  
Pears  
Oranges  
Apples  
Bananas  
Strawberries

L'uso dell'attributo rows su un oggetto Worksheet vi darà una tupla di tuple. Ciascuna delle tuple interne rappresenta una riga, e contiene gli oggetti Cell che compongono quella riga. Anche l'attributo columns dà una tupla di tuple: quelle interne contengono gli oggetti Cell che compongono le varie colonne. Nel caso di *example.xlsx*, poiché vi sono 7 righe e 3 colonne, rows ci dà una tupla di 7 tuple (ciascuna contenente 3 oggetti Cell), mentre columns dà una tupla di 3 tuple, ciascuna contenente 7 oggetti Cell). Per accedere a una particolare tupla, si può fare riferimento ad essa mediante il suo indice nella tupla più generale. Per esempio, per ottenere la tupla che rappresenta la colonna B, si usa sheet.columns[1]. Per ottenere la tupla che contiene gli oggetti Cell nella colonna A, si usa sheet.columns[0]. Una volta che si ha una tupla che rappresenta una riga o una colonna, si può ciclare sui suoi oggetti Cell e stamparne i valori.

## Cartelle di lavoro, fogli, celle

Come rapido riepilogo, ecco una rassegna di tutte le funzioni, i metodi e i tipi di dati coinvolti nella lettura di una cella da un file di un foglio di calcolo:

1. Si importa il modulo openpyxl
2. Si chiama la funzione openpyxl.load\_workbook().
3. Si ottiene un oggetto **Workbook**.
4. Si chiama il metodo get\_active\_sheet() o get\_sheet\_by\_name() della cartella di lavoro.
5. Si ottiene un oggetto Worksheet.
6. Si usano gli indici o il metodo cell() del foglio con gli argomenti per parola chiave row e column.
7. Si ottiene un oggetto Cell.
8. Si legge l'attributo value dell'oggetto Cell.

## Progetto: lettura di dati da un foglio di calcolo

Immaginiamo di aver un foglio di calcolo con i dati del Censimento degli Stati Uniti del 2010: avete il compito noioso di esaminare le sue migliaia di righe per contare la popolazione totale e il numero di cosiddetti *census tracts* (sono semplicemente aree geografiche definite specificamente per le finalità del censimento) per ciascuna contea (*county*). Ogni riga del foglio rappresenta un singolo

census tract. Chiameremo il file del foglio di calcolo *censuspopdata.xlsx*: potete scaricarlo da <http://nostarch.com/automatestuff/>. I contenuti sono come nella Figura 12.2.

	A	B	C	D	E	F
1	CensusTract	State	County	POP2010		
9837	06075010100	CA	San Francisco	3739		
9838	06075010200	CA	San Francisco	4143		
9839	06075010300	CA	San Francisco	3852		
9840	06075010400	CA	San Francisco	4545		
9841	06075010500	CA	San Francisco	2685		
9842	06075010600	CA	San Francisco	3894		
9843	06075010700	CA	San Francisco	5592		
9844	06075010800	CA	San Francisco	4578		
9845	06075010900	CA	San Francisco	4320		
9846	06075011000	CA	San Francisco	4827		
9847	06075011100	CA	San Francisco	5164		

Figura 12.2 - Il foglio di calcolo censuspopdata.xlsx.

Anche se Excel può calcolare la somma di più celle selezionate, bisogna comunque selezionare le celle per ciascuna delle oltre 3000 contee; a mano bastano pochi secondi per calcolare la popolazione di una contea, ma ripetere l'operazione per tutto il foglio di calcolo richiederebbe ore. In questo progetto, realizzerete uno script in grado di leggere dal file Excel e di calcolare statistiche per ciascuna contea nel giro di pochi secondi.

Ecco che cosa fa il vostro programma:

- legge dati dal foglio Excel;
- conta il numero dei *census tract* per ciascuna contea;
- calcola il totale della popolazione per ciascuna contea;
- stampa i risultati.

Questo significa che il vostro codice dovrà fare le cose seguenti:

- aprire e leggere le celle di un documento Excel con il modulo `openpyxl`;
- calcolare tutti i dati dei *census tract* e della popolazione e memorizzarli in una struttura di dati;
- scrivere questa struttura di dati in un file di testo con estensione `.py`, utilizzando il modulo `pprint`.

## Passo 1: leggere i dati del foglio di calcolo

Nel file Excel *censuspopdata.xlsx* vi è un solo foglio di lavoro, con il nome 'Population by Census Tract', e ciascuna riga contiene i dati di un singolo *tract*. Le colonne sono il numero del *tract* (A), la sigla dello stato (B), il nome della contea (C) e la popolazione del *tract* (D).

Aprite una nuova finestra di file editor e inserite il codice seguente. Salvate il file con il nome `readCensusExcel.py`.

```

#! python3
# readCensusExcel.py - Crea una tabella con popolazione e numero di census tract
# per ciascuna contea.
❶ import openpyxl, pprint
print('Opening workbook...')
❷ wb = openpyxl.load_workbook('censuspopdata.xlsx')
❸ sheet = wb.get_sheet_by_name('Population by Census Tract')
countyData = {}

# DA FARE: Riempire countyData con popolazione e tract di ciascuna contea.
print('Leggo le righe...')
❹ for row in range(2, sheet.get_highest_row() + 1):
    # Ogni riga del foglio di calcolo contiene i dati di un census tract.
    state = sheet['B' + str(row)].value
    county = sheet['C' + str(row)].value
    pop = sheet['D' + str(row)].value

# DA FARE: Aprire un nuovo file di testo e scrivervi i contenuti di countyData.

```

Questo codice importa il modulo `openpyxl`, nonché il modulo `pprint` che userete per stampare i dati finali della contea ❶. Poi apre il file `censuspopdata.xlsx` ❷, prende il foglio con i dati del censimento ❸ e inizia a iterare sulle sue righe ❹.

Notate che è stata creata anche una variabile `countyData`, che conterrà popolazione e numero dei *tract* calcolati per ciascuna contea. Prima di memorizzarvi alcunché, però, dovete stabilire esattamente come verranno strutturati i dati al suo interno.

## Passo 2: popolare la struttura di dati

La struttura di dati memorizzata in `countyData` sarà un dizionario con le sigle degli Stati come chiavi. Ogni sigla sarà messa in corrispondenza con un altro dizionario, le cui chiavi sono stringhe dei nomi delle contee in quello Stato. Ogni nome di contea a sua volta sarà in corrispondenza con un dizionario con due sole chiavi, `'tracts'` e `'pop'`. Queste chiavi corrispondono al numero dei *census tract* e alla popolazione per quella contea. Per esempio, il dizionario sarà simile a questo:

```

{'AK': {'Aleutians East': {'pop': 3141, 'tracts': 1},
        'Aleutians West': {'pop': 5561, 'tracts': 2},
        'Anchorage': {'pop': 291826, 'tracts': 55},
        'Bethel': {'pop': 17013, 'tracts': 3},
        'Bristol Bay': {'pop': 997, 'tracts': 1},
        --righe omesse--
}

```

Se il dizionario precedente fosse memorizzato in `countyData`, le espressioni che seguono verrebbero valutate in questo modo:

```

>>> countyData['AK']['Anchorage']['pop']
291826
>>> countyData['AK']['Anchorage']['tracts']
55

```

Più in generale, le chiavi del dizionario `countyData` saranno fatte in questo modo:

```
countyData[sigla-dello-Stato][contea]['tracts']
```

```
countyData[sigla-dello-Stato][contea]['pop']
```

Ora che sapete come verrà strutturato `countyData`, potete scrivere il codice che lo popolerà con i dati della contea. Aggiungete il codice che segue, in fondo al vostro programma:

```
#!/usr/bin/python3
# readCensusExcel.py - Crea una tabella con popolazione e numero di census tract
# per ciascuna contea.

--righi omessi--

for row in range(2, sheet.get_highest_row() + 1):
    # Ogni riga del foglio di calcolo contiene i dati di un census tract.
    state = sheet['B' + str(row)].value
    county = sheet['C' + str(row)].value
    pop = sheet['D' + str(row)].value
    # Controlla che esista la chiave per questo Stato.
    ❶ countyData.setdefault(state, {})

    # Controlla che in questo Stato esista la chiave per questa contea.
    ❷ countyData[state].setdefault(county, {'tracts': 0, 'pop': 0})

    # Ogni riga rappresenta un tract, perciò si incrementa di uno.
    ❸ countyData[state][county]['tracts'] += 1
    # Aumenta la popolazione della contea con la popolazione in questa categoria.
    ❹ countyData[state][county]['pop'] += int(pop)

# DA FARE: Aprire un nuovo file di testo e scrivervi i contenuti di countyData.
```

Le ultime due righe di codice svolgono il calcolo effettivo, incrementando il valore di `tracts` ❸ e aumentando il valore di `pop` ❹ per la contea corrente a ciascuna iterazione del ciclo `for`.

L'altro codice è qui perché non si può aggiungere il dizionario di una contea come valore per una chiave sigla di uno Stato, se la chiave stessa non esiste in `countyData`. (In altre parole, `countyData ['AK'] ['Anchorage']['tracts'] += 1` provocherà un errore, se la chiave 'AK' non esiste ancora.) Per essere sicuri che la sigla dello stato esista nella struttura di dati, bisogna chiamare il metodo `setdefault()` per impostare un valore, se non ne esiste già uno per `state` ❶.

Come il dizionario `countyData` ha bisogno di un dizionario come valore per ciascuna sigla di Stato, ciascuno di questi altri dizionari avrà bisogno di un proprio dizionario come valore per ciascuna chiave che rappresenta una contea ❷. E ciascuno di questi ulteriori dizionari a sua volta avrà bisogno di chiavi 'tracts' e 'pop' che inizino con il valore intero 0. (Se vi perdete nella struttura del dizionario, tornate all'esempio di dizionario all'inizio di questa sezione.)

Poiché `setdefault()` non farà nulla se la chiave non esiste già, potete chiamarla senza alcun problema a ogni iterazione del ciclo `for`.

## Passo 3: scrivere i risultati in un file

Una volta che il ciclo `for` si è concluso, il dizionario `countyData` conterrà tutte le informazioni su popolazione e *census tract* organizzate per contea e Stato. A questo punto, potete predisporre altro codice per scrivere il tutto in un file di testo o in un altro foglio Excel. Per il momento, usiamo semplicemente la funzione `pprint.pformat()` per scrivere il valore del dizionario `countyData` come una grande stringa in un file dal nome *census2010.py*.

Aggiungete il codice seguente in fondo al vostro programma (fate attenzione a che non sia rientrato, così da rimanere all'esterno del ciclo for):

```
#!/usr/bin/python3
# readCensusExcel.py - Crea una tabella con popolazione e numero di census tract
# per ciascuna contea.
--righe omesse--

for row in range(2, sheet.get_highest_row() + 1):
    --righe omesse--

# Apre un nuovo file di testo e vi scrive i contenuti di countyData.
print('Sto scrivendo i risultati...')
resultFile = open('census2010.py', 'w')
resultFile.write('allData = ' + pprint.pformat(countyData))
resultFile.close()
print('Fatto.')
```

La funzione `pprint.pformat()` produce una stringa che è formattata come codice Python valido. Mandandola in output a un file di testo con il nome `census2010.py`, generate un programma Python dal vostro programma Python. Può sembrare complicato, ma il vantaggio è che ora potete importate `census2010.py` come qualsiasi altro modulo Python. Nella shell interattiva, cambiate la directory di lavoro corrente in modo che sia la cartella in cui si trova il file `census2010.py` appena creato (sul mio laptop, è la cartella `C:\Python35-32`, la vostra potrebbe essere diversa, quindi sostituite l'indicazione corretta nel codice), poi importatelo:

```
>>> import os
>>> os.chdir('C:\\Python35-32')
>>> import census2010
>>> census2010.allData['AK']['Anchorage']
{'pop': 291826, 'tracts': 55}
>>> anchoragePop = census2010.allData['AK']['Anchorage']['pop']
>>> print(Nel 2010 la popolazione di Anchorage era di ' + str(anchoragePop + ' persone'))
Nel 2010 la popolazione di Anchorage era di 291826 persone
```

Il programma `readCensusExcel.py` era codice “usa e getta”: non appena avete salvato i suoi risultati in `census2010.py`, non avete più bisogno di eseguire nuovamente quel programma. Ogni volta che vi servono i dati sulle contee, basta che eseguiate `import census2010`.

Il calcolo manuale di questi dati vi avrebbe richiesto ore; questo programma ha fatto tutto in pochi secondi. Con OpenPyXL, non avrete problemi a estrarre informazioni che sono salvate in un foglio Excel e a svolgere calcoli su quelle informazioni. Potete scaricare il programma completo dall’indirizzo <http://nostarch.com/automatestuff/>.

## Idee per programmi simili

Molte aziende usano Excel per memorizzare vari tipi di dati, e non è raro che i fogli di calcolo diventino enorme e difficilmente gestibili. Qualsiasi programma che analizzi un foglio Excel ha una struttura simile: carica il file del foglio di calcolo, predisponde alcune variabili o strutture di dati, poi cicla su ciascuna delle righe nel foglio. Un programma del genere potrebbe fare cose di questo tipo:

- confrontare dati in più righe di un foglio di calcolo;

- aprire più file Excel e confrontarne i dati;
- verificare se un foglio di calcolo ha righe vuote o dati non validi in qualche cella e avvertire l’utente se si verificano queste condizioni;
- leggere dati da un foglio di calcolo e usarli quindi come input per altri programmi Python.

## Scrivere documenti Excel

OpenPyXL dà anche il modo di scrivere dati, il che significa che i vostri programmi possono creare e modificare file di foglio di calcolo. Con Python, è semplice creare fogli con migliaia di righe di dati.

## Creare e salvare documenti Excel

Chiamate la funzione `openpyxl.Workbook()` per creare un nuovo oggetto Workbook vuoto. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Foglio']
>>> sheet = wb.get_active_sheet()
>>> sheet.title
'Sheet'
>>> sheet.title = 'Spam Bacon Eggs Sheet'
>>> wb.get_sheet_names()
['Spam Bacon Eggs Sheet']
```

La cartella di lavoro si aprirà con un unico foglio con il nome Sheet (openpyxl “ragiona” in inglese). Potete cambiare il nome del foglio memorizzando una nuova stringa nel suo attributo `title`.

Ogni volta che modificate l’oggetto Workbook o i suoi fogli e le sue celle, il file non verrà salvato fino a che non chiamate il metodo `save()` della cartella di lavoro.

Inserite quanto segue nella shell interattiva (`example.xlsx` deve trovarsi nella directory di lavoro corrente):

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.title = 'Spam Spam Spam'
>>> wb.save('example_copy.xlsx')
```

Qui cambiamo il nome del nostro foglio. Per salvare le modifiche, passiamo un nome di file come stringa al metodo `save()`. Passando un nome di file diverso rispetto all’originale, per esempio ‘`example_copy.xlsx`’, si salvano le modifiche in una copia del foglio.

Ogni volta che si modifica un foglio caricato da un file, bisogna sempre salvare il nuovo foglio modificato con un nome di file diverso dall’originale. In questo modo avrete sempre il file originale con cui ricominciare a lavorare, nel caso in cui un errore nel vostro codice abbia modificato in modo errato i dati.

## Creare ed eliminare fogli

Si possono aggiungere ed eliminare fogli in una cartella di lavoro con i metodi `create_sheet()` e `remove_sheet()`. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl  
>>> wb = openpyxl.Workbook()  
>>> wb.get_sheet_names()  
['Sheet']  
>>> wb.create_sheet()  
<Worksheet "Sheet1">  
>>> wb.get_sheet_names()  
['Sheet', 'Sheet1']  
>>> wb.create_sheet(index=0, title='First Sheet')  
<Worksheet "First Sheet">  
>>> wb.get_sheet_names()  
['First Sheet', 'Sheet', 'Sheet1']  
>>> wb.create_sheet(index=2, title='Middle Sheet')  
<Worksheet "Middle Sheet">  
>>> wb.get_sheet_names()  
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
```

Il metodo `create_sheet()` restituisce un nuovo oggetto Worksheet non il nome SheetX, che per default è impostato all'ultimo foglio nella cartella di lavoro. Facoltativamente, si possono specificare indice e nome del nuovo foglio con gli argomenti per parola chiave `index` e `title`.

Continuate l'esempio precedente inserendo quanto segue:

```
>>> wb.get_sheet_names()  
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']  
>>> wb.remove_sheet(wb.get_sheet_by_name('Middle Sheet'))  
>>> wb.remove_sheet(wb.get_sheet_by_name('Sheet1'))  
>>> wb.get_sheet_names()  
['First Sheet', 'Sheet']
```

Il metodo `remove_sheet()` prende come argomento un oggetto Worksheet, non una stringa con il nome del foglio. Se conoscete solo il nome del foglio che volete eliminare, chiamate `get_sheet_by_name()` e passate a `remove_sheet()` il valore che restituisce.

Ricordatevi di chiamare il metodo `save()` per salvare le modifiche, dopo aver aggiunto o eliminato fogli dalla cartella di lavoro.

## Scrivere valori nelle celle

Scrivere valori nelle celle è un po' come scrivere valori per le chiavi di un dizionario. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl  
>>> wb = openpyxl.Workbook()  
>>> sheet = wb.get_sheet_by_name('Sheet')  
>>> sheet['A1'] = 'Hello world!'  
>>> sheet['A1'].value  
'Hello world!'
```

Se avete le coordinate della cella in forma di stringa, potete usarle come una chiave di dizionario

sull'oggetto Worksheet per specificare in quale cella volete scrivere.

## Progetto: aggiornamento di un foglio di calcolo

In questo progetto, scriverete un programma per aggiornare le celle in un foglio di calcolo in cui sono registrate le vendite di frutta e verdura. Il programma esaminerà il foglio, troverà prodotti specifici e ne aggiornerà il prezzo. Scaricate questo foglio di calcolo dall'indirizzo <http://nostarch.com/automatestuff/>. La Figura 12.3 mostra l'aspetto di una porzione del foglio.

A	B	C	D	E	F	G
1 PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL			
2 Potatoes	0,86	21,6	18,58			
3 Okra	2,26	38,6	87,24			
4 Fava beans	2,69	32,8	88,23			
5 Watermelon	0,66	27,3	18,02			
6 Garlic	1,19	4,9	5,83			
7 Parsnips	2,27	1,1	2,5			
8 Asparagus	2,49	37,9	94,37			
9 Avocados	3,23	9,2	29,72			
10 Celery	3,07	28,9	88,72			
11 Okra	2,26	40	90,4			

Figura 12.3 – Un foglio di calcolo con le vendite di prodotti ortofrutticoli.

Ciascuna riga rappresenta una singola vendita. Le colonne sono il tipo di prodotto venduto (A), il costo per libbra di quel prodotto (B), il numero di libbre vendute (C) e l'incasso totale della vendita (D). La colonna TOTAL è impostata alla formula Excel =ROUND(B3\*C3, 2), che moltiplica il costo per libbra per il numero delle libbre vendute e arrotonda il risultato al centesimo. Con questa formula, le celle nella colonna TOTAL si aggiorneranno automaticamente, se si verifica una variazione nelle colonne B o C.

Ora immaginate che il prezzo di aglio, sedano e limoni sia stato inserito in modo errato: toccherebbe a voi aggiornare il costo per libbra in tutte le righe che si riferiscono a vendite di aglio, sedano e limoni. Non potete fare un semplice “trova e sostituisci” perché potrebbero esserci altri prodotti con lo stesso prezzo e non volete “correggerli” in modo errato. Visto che le righe sono migliaia, vi ci vorrebbero ore per portare a termine il compito manualmente, ma potete scrivere un programma che lo faccia per voi in pochi secondi.

Il programma deve fare queste cose:

- ciclare su tutte le righe;
- se la riga riguarda aglio (*garlic*), sedano (*celery*) o limoni (*lemons*), modifica il prezzo.

Questo significa che il vostro codice deve fare le cose seguenti:

- aprire il foglio di calcolo;
- per ogni riga, verificare se il valore nella colonna A è Celery, Garlic O Lemon;
- se lo è, aggiornare il prezzo nella colonna B;
- salvare il foglio in un nuovo file (così da non perdere il foglio di partenza, non si sa mai).

## Passo 1: impostare una struttura di dati con le informazioni aggiornate

I prezzi che dovete aggiornare sono i seguenti:

Celery	1.19
Garlic	3.07
Lemon	1.27

Potreste scrivere il codice in questo modo:

```
if produceName == 'Celery':  
    cellObj = 1.19  
if produceName == 'Garlic':  
    cellObj = 3.07  
if produceName == 'Lemon':  
    cellObj = 1.27
```

Avere i dati su prodotti e prezzi aggiornati codificati rigidamente in questo modo è poco elegante. Se vi capitasse di dover aggiornare nuovamente il foglio con altri prezzi o per prodotti diversi, dovreste modificare gran parte del codice e, ogni volta che si modifica il codice, si rischia di introdurre qualche errore.

Una soluzione più flessibile consiste nel memorizzare le informazioni corrette sui prezzi in un dizionario e scrivere il codice in modo che usi questa struttura di dati. In una nuova finestra di file editor, inserite il codice seguente:

```
#! python3  
# updateProduce.py - Modifica i prezzi nel foglio delle vendite.  
  
import openpyxl  
  
wb = openpyxl.load_workbook('produceSales.xlsx')  
sheet = wb.get_sheet_by_name('Sheet')  
  
# I tipi di prodotti e i loro prezzi aggiornati  
PRICE_UPDATES = {'Garlic': 3.07,  
                 'Celery': 1.19,  
                 'Lemon': 1.27}  
  
# DA FARE: Ciclare sulle righe e aggiornare i prezzi.
```

Salvatelo come *updateProduce.py*. Se vi capitasse di dover aggiornare nuovamente il foglio, dovrete aggiornare solamente il dizionario PRICE\_UPDATES, senza toccare nessun'altra parte del codice.

## Passo 2: controllare tutte le righe e aggiornare i prezzi errati

La parte successiva del programma ciclerà su tutte le righe del foglio. Aggiungete il codice seguente in fondo a *updateProduce.py*:

```
#! python3
# updateProduce.py - Modifica i prezzi nel foglio delle vendite.
```

--righe omesse--

```
# Cicla su tutte le righe e aggiorna i prezzi.
❶ for rowNum in range(2, sheet.get_highest_row()): # salta la prima riga
❷     produceName = sheet.cell(row=rowNum, column=1).value
❸     if produceName in PRICE_UPDATES:
        sheet.cell(row=rowNum, column=2).value = PRICE_UPDATES[produceName]

❹ wb.save('updatedProduceSales.xlsx')
```

Cicliamo sulle righe partendo dalla 2, perché la 1 è solo l'intestazione ❶. La cella nella colonna 1 (cioè la colonna A) sarà memorizzata nella variabile `produceName` ❷. Se `produceName` esiste come chiave nel dizionario `PRICE_UPDATES` ❸, sapete che è una riga per la quale deve essere modificato il prezzo. Il prezzo corretto si troverà in `PRICE_UPDATES[produceName]`.

Notate la pulizia che introduce nel codice l'uso di `PRICE_UPDATES`. È necessario un solo enunciato `if` per ogni tipo di prodotto da aggiornare, invece di una serie di righe di codice del tipo `if produceName == 'Garlic':`; Dato che il codice usa il dizionario `PRICE_UPDATES` invece di codificare rigidamente i nomi dei prodotti e i prezzi aggiornati nel ciclo `for`, dovrete modificare solo il dizionario `PRICE_UPDATES` e non il codice, qualora si rendano necessarie altre modifiche ai prezzi nel foglio.

Dopo aver esplorato tutto il foglio e aver apportato le modifiche, il codice salva l'oggetto `Workbook` in `updatedProduceSales.xlsx` ❹. Non sovrascrive il vecchio foglio, nell'eventualità che sia stato commesso qualche errore. Dopo aver verificato che il foglio aggiornato è corretto, potete eliminare il vecchio foglio di calcolo.

Potete scaricare il codice sorgente completo per questo programma da <http://nostarch.com/automatestuff/>.

## Idee per programmi simili

Molti usano regolarmente i fogli di calcolo Excel, perciò un programma in grado di modificare automaticamente e scrivere file di questo tipo può essere davvero utile.

Ecco alcune cose che un programma del genere potrebbe fare:

- leggere dati da un foglio e scriverli in parti di altri fogli di calcolo;
- leggere dati da siti web, file di testo o dagli Appunti e scriverli in un foglio di calcolo;
- “ripulire” automaticamente i dati nei fogli di calcolo. Per esempio, potrebbe usare espressioni regolari per leggere i diversi formati dei numeri telefonici e modificarli in modo che rispettino tutti un unico formato standard.

## Impostare lo stile dei caratteri delle celle

Modificare lo stile dei caratteri di celle, righe o colonne può contribuire a evidenziare aree importanti di un foglio di calcolo. Nel caso del foglio con le vendite dei prodotti ortofrutticoli, per esempio, il programma potrebbe mettere in grassetto le righe che si riferiscono a patate, aglio e pastinaca (*parsnip*); oppure formattare in corsivo ogni riga in cui il prezzo per libbra superi 5.00. Formattare manualmente parti di un foglio di calcolo molto grande sarebbe noioso, ma il vostro programma può farlo in un istante.

Per personalizzare gli stili dei caratteri nelle celle, dovete importare le funzioni `Font()` e `Style()` dal modulo `openpyxl.styles`.

```
From openpyxl.styles import Font, Style
```

Questo vi consente di scrive semplicemente `Font()` invece che `openpyxl.styles.Font()`. (Vedete il paragrafo “Importazione dei moduli” a [pagina 57](#) per ripassare questa forma dell’enunciato `import`.)

Ecco un esempio che crea una nuova cartella di lavoro e imposta la cella A1 in modo che i suoi caratteri siano in una font corsiva a 24 punti. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl
>>> from openpyxl.styles import Font, Style
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')
❶ >>> italic24Font = Font(size=24, italic=True)
❷ >>> styleObj = Style(font=italic24Font)
❸ >>> sheet['A1'].style = styleObj
>>> sheet['A1'] = 'Hello world!'
>>> wb.save('styled.xlsx'
```

OpenPyXL rappresenta l’insieme delle impostazioni per una cella con un oggetto `Style`, che viene memorizzato nell’attributo `style` dell’oggetto `Cell`. Si può impostare lo stile di una cella assegnando l’oggetto `Style` all’attributo `style`.

In questo esempio, `Font(size=24, italic=True)` restituisce un oggetto `Font`, che è memorizzato in `italic24Font` ❶. Gli argomenti per parola chiave di `Font()`, `size` e `italic`, configurano gli attributi di stile dell’oggetto `Font`. Questo oggetto `Font` poi viene passato alla chiamata `Style(font=italic24Font)`, che restituisce il valore memorizzato in `styleObj` ❷. Quando `styleObj` viene assegnato all’attributo `style` della cella ❸, tutte quelle informazioni di stile vengono applicate alla cella A1.

## Oggetti Font

Gli attributi `style` negli oggetti `Font` influenzano il modo in cui viene visualizzato il testo nelle celle. Per impostare gli attributi di stile per i caratteri, si passano argomenti per parola chiave a `Font()`. La [Tabella 12.2](#) elenca i possibili argomenti per parola chiave della funzione `Font()`.

**Tabella 12.2** – Argomenti per parola chiave per gli attributi `style` dei caratteri.

Argomento per parola chiave	Tipo di dati	Descrizione
<code>name</code>	stringa	Il nome del tipo di carattere, come ‘Calibri’ o ‘Times New Roman’
<code>size</code>	intero	Il corpo, in punti
<code>bold</code>	Booleano	True, per i caratteri in grassetto ( <i>bold</i> )
<code>italic</code>	Booleano	True, per i caratteri corsivi ( <i>italic</i> )

Potete chiamare `Font()` per creare un oggetto `Font` e memorizzarlo in una variabile. Poi lo passate a `Style()`, memorizzate l’oggetto `Style` risultante in una variabile e assegnate quella variabile all’attributo

style di un oggetto Cell. Per esempio, questo codice crea vari stili di carattere:

```
>>> import openpyxl  
>>> from openpyxl.styles import Font, Style  
>>> wb = openpyxl.Workbook()  
>>> sheet = wb.get_sheet_by_name('Sheet')  
  
>>> fontObj1 = Font(name='Times New Roman', bold=True)  
>>> styleObj1 = Style(font=fontObj1)  
>>> sheet['A1'].style = styleObj1  
>>> sheet['A1'] = 'Bold Times New Roman'  
  
>>> fontObj2 = Font(size=24, italic=True)  
>>> styleObj2 = Style(font=fontObj2)  
>>> sheet['B3'].style = styleObj2  
>>> sheet['B3'] = '24 pt Italic'  
  
>>> wb.save('styles.xlsx')
```

Qui, memorizziamo in fontObj1 un oggetto Font e lo usiamo per creare un oggetto Style, che memorizziamo in styleObj1, poi impostiamo l'attributo style dell'oggetto Cell per la cella A1 a styleObj1. Ripetiamo il procedimento con un altro oggetto Font e un altro oggetto Style per impostare lo stile di una seconda cella. Dopo aver eseguito questo codice, gli stili delle celle A1 e B3 nel foglio di lavoro saranno personalizzati, come nella [Figura 12.4](#).

	A	B	C	D
1	<b>Bold Times New Roman</b>			
2				
3		<i>24 pt Italic</i>		
4				
5				

**Figura 12.4** – Un foglio di calcolo con stili di carattere personalizzati.

Per la cella A1 impostiamo il nome della font a 'Times New Roman' e impostiamo bold a True, in modo che il testo appaia in Times New Roman grassetto. Non abbiamo specificato una dimensione, perciò viene utilizzato il valore predefinito di openpyxl, 11.

Nella cella B3 abbiamo impostato il testo a corsivo 24 punti; non abbiamo specificato il nome di una font, perciò è stato utilizzato il tipo di carattere di default di openpyxl, Calibri.

## Formule

Le formule, che iniziano con un segno di uguale, possono configurare le celle in modo che contengano valori calcolati a partire da altre celle. In questa sezione, userete il modulo openpyxl per aggiungere formule alle celle da programma, come fareste con valori normali. Per esempio:

```
>>> sheet['B9'] = '=SUM(B1:B8)'
```

Questo memorizzerà =SUM(B1:B8) come valore nella cella B9. La cella B9 quindi conterrà una formula che calcola la somma dei valori nelle celle da B1 a B8. Potete vedere la formula in azione nella

Figura 12.5 (la formula viene tradotta automaticamente nella versione italiana).

	A	B	C	D	E	F
1		82				
2		11				
3		85				
4		18				
5		57				
6		51				
7		38				
8		42				
9	TOTALE:	384				
10						
11						

Figura 12.5 – La cella B9 contiene la formula =SOMMA(B1:B8), che somma le celle da B1 a B8.

Una formula si impone come qualsiasi altro valore testuale in una cella. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl  
>>> wb = openpyxl.Workbook()  
>>> sheet = wb.get_active_sheet()  
>>> sheet['A1'] = 200  
>>> sheet['A2'] = 300  
>>> sheet['A3'] = '=SUM(A1:A2)'  
>>> wb.save('writeFormula.xlsx')
```

Le celle A1 e A2 conterranno, rispettivamente, 200 e 300. Il valore della cella A3 è impostato a una formula che somma i valori in A1 e A2. Quando si apre il foglio in Excel, A3 visualizzerà il valore 500. Si può leggere la formula contenuta in una cella come si farebbe per qualsiasi valore. Se però volete vedere il risultato del calcolo della formula, anziché la formula stessa, dovete passare True per l'argomento per parola chiave `data_only` a `load_workbook()`. Questo significa che un oggetto `Workbook` può mostrare o le formule o il risultato delle formule, ma non entrambi. (Ma potete caricare più oggetti `Workbook` per lo stesso file di foglio di calcolo.) Inserite quanto segue nella shell interattiva per vedere la differenza fra caricare una cartella di lavoro con e senza l'argomento per parola chiave `data_only`:

```
>>> import openpyxl  
>>> wbFormulas = openpyxl.load_workbook('writeFormula.xlsx')  
>>> sheet = wbFormulas.get_active_sheet()  
>>> sheet['A3'].value  
'=SUM(A1:A2)'  
>>> wbDataOnly = openpyxl.load_workbook('writeFormula.xlsx', data_only=True)  
>>> sheet = wbDataOnly.get_active_sheet()  
>>> sheet['A3'].value  
500
```

Qui, quando si chiama `load_workbook()` con `data_only=True`, la cella A3 mostra il valore 500, il risultato della

formula =SUM(A1:A2), anziché il testo della formula.

Le formule di Excel permettono un certo grado di programmabilità dei fogli di calcolo, ma diventano rapidamente ingestibili, se le attività si fanno complicate. Per esempio, anche se conoscete bene le formule di Excel, vi verrà il mal di testa cercando di decifrare che cosa faccia realmente una formula come =IFERROR(TRIM(IF(LEN(VLOOKUP(F7, Sheet2!\$A\$1:\$B\$10000, 2, FALSE))>0,SUBSTITUTE(VLOOKUP(F7, Sheet2!\$A\$1:\$B\$10000, 2, FALSE)," ", ""),"")),""). Il codice Python è molto più leggibile.

## Regolare righe e colonne

In Excel, regolare le dimensioni di righe e colonne è facilissimo: basta fare un clic e trascinare i bordi di un'intestazione di riga o colonna. Se però dovete impostare le dimensioni di una riga o di una colonna sulla base dei contenuti delle sue celle o se volete impostare le dimensioni in un gran numero di file di fogli di calcolo, farete molto più in fretta scrivendo un programma Python che svolga quel compito per voi.

Le righe e le colonne possono anche essere nascoste, oppure possono essere “congelate” in modo da risultare sempre visibili sullo schermo e da apparire su ogni pagina quando il foglio viene stampato (il che è molto comodo per le intestazioni).

## Impostare altezza di riga e larghezza di colonna

Gli oggetti Worksheet hanno attributi `row_dimensions` e `column_dimensions` che controllano l'altezza delle righe e la larghezza delle colonne. Inserite nella shell interattiva:

```
>>> import openpyxl  
>>> wb = openpyxl.Workbook()  
>>> sheet = wb.get_active_sheet()  
>>> sheet['A1'] = 'Riga più alta'  
>>> sheet['B2'] = 'Colonna più larga'  
>>> sheet.row_dimensions[1].height = 70  
>>> sheet.column_dimensions['B'].width = 20  
>>> wb.save('dimensions.xlsx')
```

Gli attributi `row_dimensions` e `column_dimensions` di un foglio sono valori simili a dizionari: `row_dimensions` contiene oggetti `RowDimension` e `column_dimensions` contiene oggetti `ColumnDimension`. In `row_dimensions`, potete accedere a uno degli oggetti utilizzando il numero della riga (in questo caso, 1 o 2). In `column_dimensions`, potete accedere a uno degli oggetti utilizzando la lettera della colonna (in questo caso, A o B).

Il foglio `dimensions.xlsx` si presenta come nella [Figura 12.6](#).

A	B	C
1 Riga più alta		
2 Colonna più larga		

**Figura 12.6** - La riga 1 e la colonna B sono impostate a un'altezza e a una larghezza maggiori.

Una volta che avete l'oggetto `RowDimension`, potete stabilirne l'altezza. Una volta che avete l'oggetto `ColumnDimension`, potete impostarne la larghezza. L'altezza di una riga può essere un valore intero o in virgola mobile compreso fra 0 e 409, valore che rappresenta l'altezza misurata in punti, dove un punto è pari a 1/72-esimo di pollice. L'altezza di default delle righe è 12.75. La larghezza delle colonne può essere impostata a un valore intero o in virgola mobile compreso fra 0 e 255. Questo valore rappresenta il numero dei caratteri, nella dimensione di default (11 punti) visualizzabili nella cella. La larghezza di default delle colonne è 8.43 caratteri. Le colonne con larghezza 0 o le righe con altezza 0 sono nascoste.

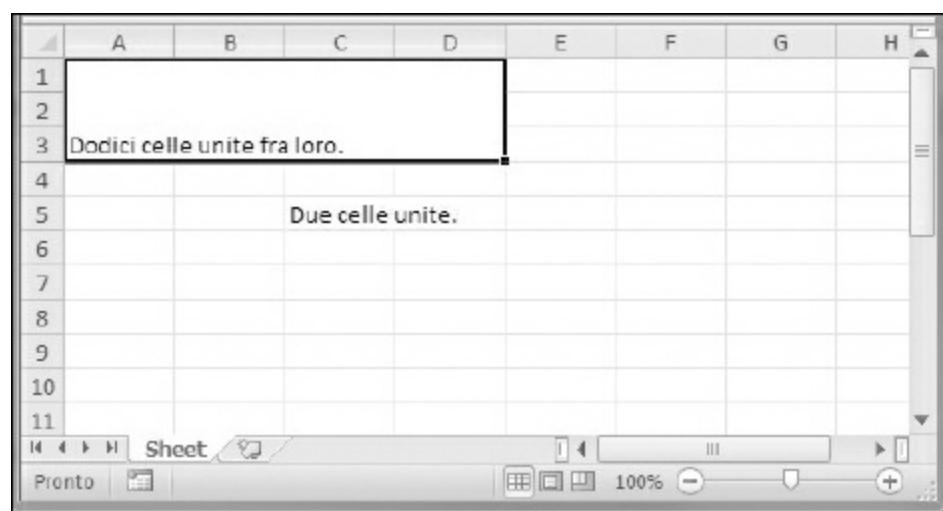
## Unire e separare celle

Si può unire in un'unica cella un'area rettangolare di celle, con il metodo del foglio `merge_cells()`. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl  
>>> wb = openpyxl.Workbook()  
>>> sheet = wb.get_active_sheet()  
>>> sheet.merge_cells('A1:D3')  
>>> sheet['A1'] = 'Dodici celle unite fra loro.'  
>>> sheet.merge_cells('C5:D5')  
>>> sheet['C5'] = 'Due celle unite.'  
>>> wb.save('merged.xlsx')
```

L'argomento di `merge_cells()` è una singola stringa contenente l'indicazione delle celle superiore sinistra e inferiore destra dell'area rettangolare da unire: 'A1:D3' fonde 12 celle in un'unica cella. Per impostare il valore di queste celle unite, basta fissare il valore della cella superiore sinistra del gruppo unito.

Quando si esegue questo codice, `merged.xlsx` dà il risultato visibile nella [Figura 12.7](#).



**Figura 12.7** - Celle unite in un foglio di calcolo.

Per separare le celle, si chiama il metodo `unmerge_cells()` del foglio. Inserite quanto segue nella shell interattiva:

```
>>> import openpyxl  
>>> wb = openpyxl.load_workbook('merged.xlsx')
```

```
>>> sheet = wb.get_active_sheet()
>>> sheet.unmerge_cells('A1:D3')
>>> sheet.unmerge_cells('C5:D5')
>>> wb.save('merged.xlsx')
```

Se salvate le modifiche e poi provate ad aprire il foglio di calcolo, vedrete che le celle unite sono tornate a essere celle singole.

## Congelare i riquadri

Quando si ha un foglio troppo grande perché possa essere visualizzato completamente, è utile “congelare” qualcuna delle righe superiori o delle colonne più a sinistra, in modo che rimangano sempre visibili, anche quando si scorre lungo le righe e le colonne del foglio. Si parla, in questo caso, di **pannelli congelati o bloccati**. In OpenPyXL, ogni oggetto Worksheet ha un attributo `freeze_panes` che può essere impostato a un oggetto `Cell` o a una stringa con le coordinate di una cella. Notate che tutte le righe al di sopra di questa cella e tutte le colonne alla sua sinistra saranno bloccate, mentre non saranno bloccate la riga e la colonna della cella stessa.

Per sbloccare tutto, impostate `freeze_panes` a `None` oppure ad '`A1`'. La [Tabella 12.3](#) mostra quali righe e quali colonne saranno bloccate per alcune impostazioni esemplificative di `freeze_panes`.

**Tabella 12.3** - Esempi di righe e colonne bloccate.

Impostazione di <code>freeze_panes</code>	Righe e colonne bloccate
<code>sheet.freeze_panes = 'A2'</code>	Riga 1
<code>sheet.freeze_panes = 'B1'</code>	Colonna Z
<code>sheet.freeze_panes = 'C1'</code>	Le colonne A e B
<code>sheet.freeze_panes = 'C2'</code>	La riga 1 e le colonne A e B
<code>sheet.freeze_panes = 'A1'</code> oppure <code>sheet.freeze_panes = None</code>	Nessuna colonna o riga bloccata

Controllate di avere ancora a disposizione il foglio con le vendite di prodotti ortofrutticoli scaricato da <http://nostarch.com/automatestuff/>. Poi inserite quanto segue nella shell interattiva.

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('produceSales.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.freeze_panes = 'A2'
>>> wb.save('freezeExample.xlsx')
```

Se si imposta l’attributo `freeze_panes` a '`A2`', la riga 1 rimarrà sempre visibile, indipendentemente da quanto l’utente scorra il foglio, come si può vedere nella [Figura 12.8](#).

	A	B	C	D	E	F
1	PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL		
3050	Apricots	3,71	3,7	13,73		
3051	Lime	1,06	7,7	8,16		
3052	Green peppers	1,89	25,8	48,76		
3053	Carrots	1,26	5,2	6,55		
3054	Cucumber	1,07	32	34,24		
3055	Coconuts	1,18	22,6	26,67		
3056	Daikon	1,4	34,9	48,86		
3057	Papaya	1,34	29,2	39,13		
3058	Celery	3,07	10,8	33,16		
3059	Bok choy	1,42	0,7	0,99		

Figura 12.8 - Con freeze\_panes impostato ad ‘A2’, la riga 1 rimane sempre visibile, anche quando l’utente scorre verso il basso.

## Grafici

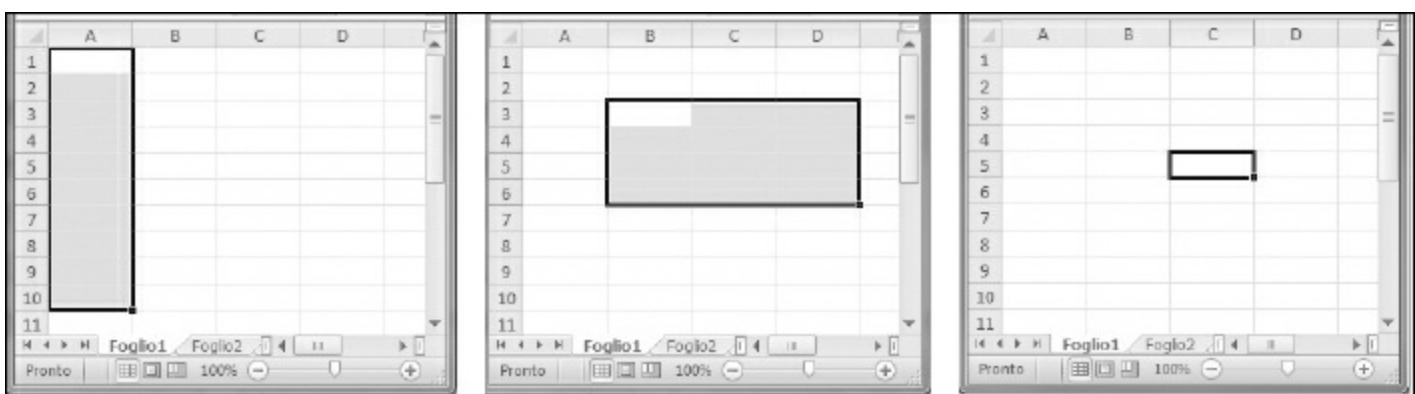
OpenPyXL permette la creazione di grafici a barre, a linee, a dispersione e a torta a partire dai dati memorizzati nelle celle di un foglio. Per creare un grafico, dovete:

1. creare un oggetto Reference a partire da una selezione rettangolare di celle;
2. creare un oggetto Series passandolo nell’oggetto Reference;
3. creare un oggetto Chart;
4. accodare l’oggetto Series all’oggetto Chart;
5. aggiungere l’oggetto Chart all’oggetto Worksheet.

L’oggetto Reference richiede qualche spiegazione. Gli oggetti Reference vengono creati chiamando la funzione `openpyxl.charts.Reference()` e passando tre argomenti:

1. l’oggetto Worksheet che contiene i dati del grafico;
2. una tupla di due interi, che rappresentano la cella superiore sinistra della selezione rettangolare di celle che contengono i dati di grafico, dove il primo intero della tupla è la riga, il secondo la colonna (notate che la prima riga è 1, non 0);
3. una tupla di due interi, che rappresentano la cella inferiore destra della selezione rettangolare di celle che contengono i dati del grafico, dove il primo intero della tupla è la riga, il secondo la colonna.

La [Figura 12.9](#) mostra alcuni esempi di argomenti coordinate.



**Figura 12.9** - Da sinistra a destra: (1, 1), (10, 1); (3, 2), (6, 4); (5, 3), (5, 3).

Inserite questo esempio nella shell interattiva, per creare un grafico a barre e aggiungetelo al foglio di calcolo:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_active_sheet()
>>> for i in range(1, 11):                      # crea dei dati nella colonna A
    sheet['A' + str(i)] = i

>>> ref0bj = openpyxl.chart.Reference(sheet, min_col=1, min_row=1, max_col=1, max_row=10)

>>> series0bj = openpyxl.chart.Series(ref0bj, title='Prima serie')

>>> chart0bj = openpyxl.chart.BarChart()
>>> chart0bj.title = 'My Chart'
>>> chart0bj.append(series0bj)

>>> sheet.add_chart(chart0bj, 'C5')
>>> wb.save('sampleChart.xlsx')
```

Questo produce un foglio il cui aspetto è come nella [Figura 12.10](#).

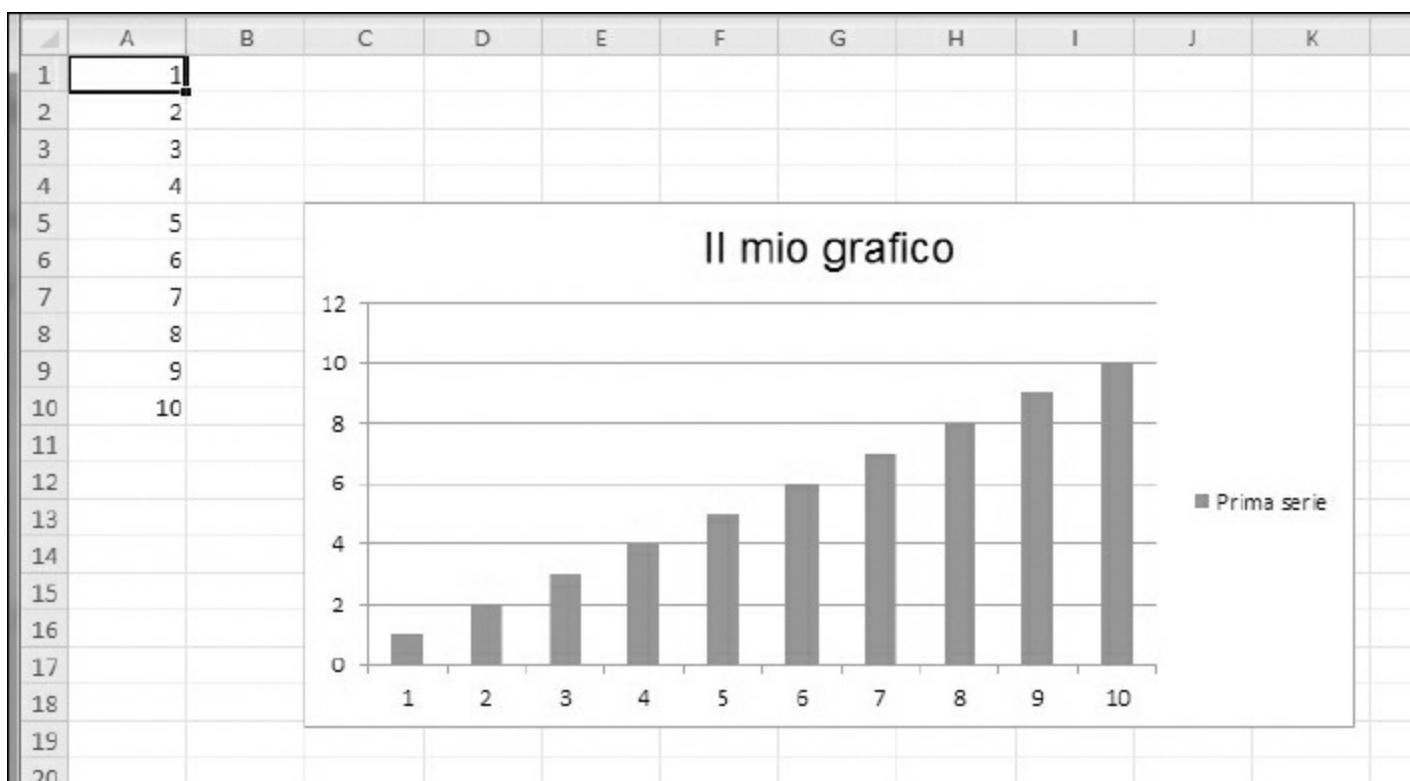


Figura 12.10 - Un foglio a cui è stato aggiunto un grafico, il vertice superiore sinistro è in CS.

Abbiamo così creato un grafico a barre chiamando `openpyxl.chart.BarChart()`. Potete creare anche grafici a linee, a dispersione e a torta chiamando invece, rispettivamente, `openpyxl.chart.LineChart()`, `openpyxl.chart.ScatterChart()`, e `openpyxl.chart.PieChart()`.

Purtroppo, nella versione attuale di OpenPyXL, la funzione `load_workbook()` non carica i grafici già presenti nei file Excel. Anche se il file Excel contiene dei grafici, l'oggetto `Workbook` caricato non li include.

Se caricate un oggetto `Workbook` e lo salvate immediatamente con lo stesso nome di file `.xlsx`, eliminate tutti i grafici che contiene.

## Riepilogo

Spesso la parte difficile, nell'elaborazione delle informazioni, non è l'elaborazione stessa, ma semplicemente il **predisporre i dati nel formato giusto** per il programma. Una volta caricato un foglio di calcolo in Python, però, potete estrarre e manipolare i dati molto più rapidamente di quel che potreste fare a mano.

Potete anche **generare fogli di calcolo** come output dei vostri programmi. Così, se un collega ha bisogno di trasferire in un foglio di calcolo un file di testo o un PDF con migliaia di contatti, per esempio, non sarà necessario copiarli e incollarli tutti in Excel.

Armati del modulo `openpyxl` e di un po' di conoscenza di programmazione, manipolare anche il più grande dei fogli di calcolo sarà una passeggiata.

## Domande di ripasso

Per le domande seguenti, immaginate di avere un oggetto `Workbook` nella variabile `wb`, un oggetto `Worksheet` in `sheet`, un oggetto `Cell` in `cell`, un oggetto `Comment` in `comm` e un oggetto `Image` in `img`.

1. Che cosa restituisce la funzione `openpyxl.load_workbook()`?

2. Che cosa restituisce il metodo `get_sheet_names()` dell'oggetto `Workbook`?
3. Come recuperereste l'oggetto `Worksheet` per un foglio che si chiama 'Foglio1'?
4. Come recuperereste l'oggetto `Worksheet` per il foglio attivo della cartella di lavoro?
5. Come recuperereste il valore nella cella C5?
6. Come impostereste a "Hello" il contenuto della cella C5?
7. Come recuperereste riga e colonna di una cella in forma di interi?
8. Che cosa restituiscono i metodi `get_highest_column()` e `get_highest_row()` del foglio di lavoro e qual è il tipo di dati di questi valori di ritorno?
9. Se avete bisogno dell'indice della colonna 'M' sotto forma di intero, quale funzione dovreste chiamare?
10. Se avete bisogno del nome della colonna 14 in forma di stringa, quale funzione dovreste chiamare?
11. Come potete recuperare una tupla con tutti gli oggetti `Cell` da A1 a F1?
12. Come salvereste la cartella di lavoro con il nome di file `example.xlsx`?
13. Come impostereste una formula in una cella?
14. Volete recuperare il risultato della formula in una cella, anziché la formula stessa. Che cosa dovete fare innanzitutto?
15. Come impostereste l'altezza della riga 5 a 100?
16. Come nascondereste la colonna C?
17. Indicate alcune caratteristiche che OpenPyXL 2.4.2 non carica da un file di foglio di calcolo.
18. Che cos'è un pannello bloccato
19. ?
20. Quali sono le funzioni e i metodi che bisogna chiamare per creare un grafico a barre?

## Un po' di pratica

Per esercitarvi, scrivete dei programmi che svolgano queste attività.

### Tabelline per la moltiplicazione

Create un programma `multiplicationTable.py` che prenda un numero  $N$  dalla riga di comando e crei una tabellina di moltiplicazione  $N \times N$  in un foglio Excel. Per esempio, quando il programma viene eseguito in questo modo:

```
py multiplicationTable.py 6
```

deve creare un foglio analogo a quello della [Figura 12-11](#).

	A	B	C	D	E	F	G
1		1	2	3	4	5	6
2	1	1	2	3	4	5	6
3	2	2	4	6	8	10	12
4	3	3	6	9	12	15	18
5	4	4	8	12	16	20	24
6	5	5	10	15	20	25	30
7	6	6	12	18	24	30	36
8							

Figura 12.11 - Una tabellina della moltiplicazione generata in un foglio Excel.

La riga 1 e la colonna A devono essere utilizzate per le etichette e devono essere in grassetto.

## Inserimento di righe vuote

Create un programma *blankRowInserter.py*, che prenda due interi e una stringa con un nome di file come argomenti da riga di comando. Chiamiamo  $N$  il primo intero,  $M$  il secondo. A partire dalla riga  $N$ , il programma deve inserire  $M$  righe vuote nel foglio. Per esempio, quando il programma viene eseguito in questo modo:

```
python blankRowInserter.py 3 2 myProduce.xlsx
```

I fogli “prima” e “dopo” devono essere come nella Figura 12.12.

A	B	C	D	E	F
1 Patate	Sedano	Insalata	Melanzane	Carote	Pesche
2 Aglio	Zenzero	Pomodori	Rape	Ravanelli	Noci
3 Cavoli	Mele	Peperoni	Pompelmi	Cocco	Nocciole
4 Insalata	Arance	Sedano	Limoni	Bietole	Mandorle
5 Pomodori	Cocco	Zenzero	Arance	Ananas	Fichi
6 Peperoni	Ananas	Zucchine	Cipolle	Cipolle	Patate
7 Zucchine	Pere	Melanzane	Aglio	Rape	Prugne
8 Melanzane	Spinaci	Cipolle	Sedano	Zucchine	Meloni
9 Rape	Bietole	Mandarini	Topinambur	Ciliegie	Zucchine

A	B	C	D	E	F
1 Patate	Sedano	Insalata	Melanzane	Carote	Pesche
2 Aglio	Zenzero	Pomodori	Rape	Ravanelli	Noci
3					
4					
5 Cavoli	Mele	Peperoni	Pompelmi	Cocco	Nocciole
6 Insalata	Arance	Sedano	Limoni	Bietole	Mandorle
7 Pomodori	Cocco	Zenzero	Arance	Ananas	Fichi
8 Peperoni	Ananas	Zucchine	Cipolle	Cipolle	Patate
9 Zucchine	Pere	Melanzane	Aglio	Rape	Prugne

Figura 12.12 - Un foglio prima (a sinistra) e dopo (a destra) l’inserimento di due righe vuote a partire dalla riga 3.

Potete scrivere questo programma leggendo i contenuti del foglio. Poi, nello scriverli in un nuovo foglio, usate un ciclo **for** per copiare le prime  $N$  righe. Per le righe rimanenti, sommate  $M$  al numero di riga nel foglio di output.

## Invertitore di celle

Scrivete un programma che inverta riga e colonna delle celle in un foglio. Per esempio, il valore che si trova alla riga 5, colonna 3, dovrà poi trovarsi nella riga 3, colonna 5 (e viceversa). Questo per tutte le celle nel foglio. Per esempio, un foglio “prima” e “dopo” dovrà presentarsi simile a quello che si vede nella Figura 12.13.

	A	B	C	D	E	F	G
1	PRODOTTI	VENDUTI					
2	Patate	50					
3	Cavoli	63					
4	Insalata	75					
5	Pomodori	21					
6	Peperoni	48					
7	Zucchine	95					
8	Melanzane	113					
9	Rape	18					
10							

	A	B	C	D	E	F	G	H
1	PRODOTTI	Patate	Cavoli	Insalata	Pomodori	Peperoni	Zucchine	Melan...
2	VENDUTI	50	63	75	21	48	95	
3								
4								
5								
6								
7								
8								
9								
10								

**Figura 12.13** - Un foglio di calcolo prima (sopra) e dopo (sotto) l'inversione.

Potete scrivere questo programma utilizzando cicli for annidati per leggere i dati del foglio in una struttura di dati che sia una lista di liste. Questa struttura potrebbe avere `sheetData[x][y]` per la cella all'incrocio della colonna  $x$  e della riga  $y$ . Poi, quando scrivete il nuovo foglio, usate `sheetData[y][x]` per spostare la cella all'incrocio della colonna  $x$  e della riga  $y$ .

## Da file di testo a foglio di calcolo

Scrivete un programma che legga i contenuti di vari file di testo (potete crearli direttamente voi) e inserisca quei contenuti in un foglio di calcolo, con una riga di testo per riga. Le righe del primo file di testo saranno nelle celle della colonna A, quelle del secondo file di testo nelle celle della colonna B e via di seguito.

Usate il metodo `readlines()` dell'oggetto `File` per restituire una lista di stringhe, una stringa per ciascuna riga nel file.

Per il primo file, mandate la prima riga alla colonna 1, riga 1 del foglio.

La seconda riga di testo andrà scritta nella colonna 1, riga 2 del foglio e così via.

Il file successivo che verrà letto con `readlines()` verrà scritto nella colonna 2; quello successivo ancora nella colonna 3 e via di questo passo.

## Da foglio di calcolo a file di testo

Scrivete un programma che svolga l'attività contraria rispetto al programma precedente: dovrà aprire un foglio di calcolo e scrivere le celle della colonna A in un file di testo, le celle della colonna B in

un altro file di testo e così via.

# Lavorare con documenti PDF e Word

I documenti PDF e Word sono **file binari**, il che li rende molto più complessi dei file di puro di testo. Oltre al testo, memorizzano grandi quantità di informazioni su **caratteri, colori, impaginazione**. Se volete che i vostri programmi leggano o scrivano documenti PDF o Word, dovete fare molto più che passare i loro nomi di file a `open()`.

Per fortuna, esistono moduli Python che semplificano l'interazione con questi tipi di documenti e questo capitolo ne esaminerà due, PyPDF2 e Python-Docx.

## Documenti PDF

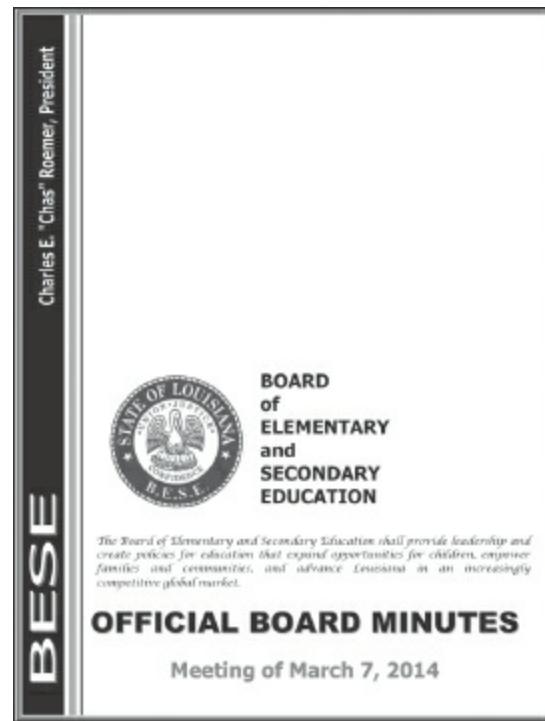
PDF è l'acronimo di **Portable Document Format** e i relativi documenti usano l'estensione di file `.pdf`. I PDF offrono molte caratteristiche, ma in questo capitolo ci concentreremo sulle due cose che capita più spesso di dover fare con questo tipo di documenti: **leggerne** i contenuti testuali e **creare** nuovi **PDF** da documenti esistenti. Il modulo che userete per lavorare con i PDF è PyPDF2, Per installarlo, eseguite `pip install PyPDF2` dalla riga di comando. Il nome di questo modulo è sensibile alla differenza fra maiuscole e minuscole, perciò fate attenzione: la `y` deve essere minuscola, tutto il resto maiuscolo. L'[Appendice A](#) spiega in dettaglio come installare i moduli di terze parti.  
Se il modulo è stato installato correttamente, l'esecuzione di `import PyPDF2` nella shell interattiva non deve generare alcun errore.

### Il problematico formato PDF

I file PDF sono eccellenti per facilitare la stampa e la lettura da parte degli esseri umani, ma la loro analisi per la trasformazione in puro testo con un programma è tutt'altro che immediata. PyPDF2 potrebbe commettere degli errori nell'estrarrre il testo da un PDF e in qualche caso potrebbe addirittura non riuscire ad aprire qualche PDF. Purtroppo non c'è molto che si possa fare: semplicemente, bisogna prendere atto che PyPDF2 non è in grado di lavorare con qualche particolare file PDF. Detto questo, devo

## Estrarre testo da PDF

PyPDF2 non ha modo di estrarre dai documenti PDF immagini, grafici o altri elementi mediiali, ma può estrarre il testo e restituirlo come una stringa Python. Per iniziare ad apprendere come funziona PyPDF2, lo useremo sul PDF di esempio ([Figura 13.1](#)).



**Figura 13.1** - La pagina PDF da cui estrarremo il testo.

Scaricate questo PDF dall'indirizzo <http://nostarch.com/automatestuff/> e inserite quanto segue nella shell interattiva:

```
>>> import PyPDF2
>>> pdfFileObj = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
❶ >>> pdfReader.numPages
19
❷ >>> pageObj = pdfReader.getPage(0)
❸ >>> pageObj.extractText()
'OOFFFFIICCIALL BBOOAARRDD MMIINNUUTTEESS Meeting of March 7, 2015
\n The Board of Elementary and Secondary Education shall provide leadership
and create policies for education that expand opportunities for children,
empower families and communities, and advance Louisiana in an increasingly
competitive global market. BOARD OF ELEMENTARY and SECONDARY EDUCATION '
```

Come prima cosa, viene importato il modulo PyPDF2. Poi si apre mettingminutes.pdf in modalità lettura binaria e lo si memorizza in pdfFileObj. Per avere un oggetto PdfFileReader che rappresenti questo PDF, si chiama PyPDF2.PdfFileReader() e vi si passa pdfFileObj. Si memorizza poi questo oggetto PdfFileReader in pdfReader.

Il numero totale delle pagine nel documento è memorizzato nell'attributo numPages di un oggetto PdfFileReader **❶**. Il PDF di esempio è costituito da 19 pagine, ma qui estraiamo il testo solo dalla prima

pagina.

Per estrarre il testo da una pagina, bisogna ottenere un oggetto `Page`, che rappresenta una singola pagina di un PDF, da un oggetto `PdfFileReader`. Si può ottenere un oggetto `Page` chiamando il metodo `getPage()` ❷ su un oggetto `PdfFileReader`, passandogli il numero di pagina della pagina a cui si è interessati – in questo caso, 0.

PyPDF2 usa un indice a base zero per individuare le pagine: la prima pagina è la pagina 0, la seconda è la pagina 1 e così via, e questo sempre, comunque siano numerate le pagine nel documento. Se per esempio il vostro PDF fosse un estratto di tre pagine ricavato da una relazione più lunga, e le sue pagine portassero i numeri 42, 43 e 44, per ottenere la prima pagina del documento dovreste comunque chiamare `pdfReader.getPage(0)` e non `getPage(42)` o `getPage(1)`.

Una volta ottenuto un oggetto `Page`, chiamate il suo metodo `extractText()` per avere di ritorno una stringa con il testo della pagina ❸. L'estrazione del testo non è perfetta: il testo *Charles E. "Chas" Roemer, President*, presente nel PDF, non è presente invece nella stringa restituita da `extractText()` e ogni tanto gli spazi sono saltati. Comunque, questa approssimazione del contenuto testuale del PDF può essere sufficiente per le finalità del nostro programma.

## Decifrare i PDF

Alcuni documenti PDF sono cifrati, il che ne impedisce la lettura se chi apre il documento non fornisce la password giusta. Inserite quanto segue nella shell interattiva con il PDF che avete scaricato, che è stato cifrato con la password `rosebud`:

```
>>> import PyPDF2
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
◆ >>> pdfReader.isEncrypted
True
❖ >>> pdfReader.getPage(0)
Traceback (most recent call last):
File "<pyshell#173>", line 1, in <module>
    pdfReader.getPage()
--rige omesse--
File "C:\Python34\lib\site-packages\PyPDF2\pdf.py", line 1173, in getObjet
    raise utils.PdfReadError("file has not been decrypted")
PyPDF2.utils.PdfReadError: file has not been decrypted
• >>> pdfReader.decrypt('rosebud')
1
>>> page0obj = pdfReader.getPage(0)
```

Tutti gli oggetti `PdfFileReader` hanno un attributo `isEncrypted` che ha valore `True` se il PDF è cifrato e `False` altrimenti ❶. Qualsiasi tentativo di chiamare una funzione che legga il file prima che questo sia stato decifrato con la password corretta darà come risultato un errore ❷.

Per leggere un PDF cifrato, si chiama prima la funzione `decrypt()` e le si passa la password come stringa ❸. Dopo aver chiamato `decrypt()` con la password corretta, si vede che la chiamata di `getPage()` non provoca più un errore. Se le si dà una password sbagliata, la funzione `decrypt()` restituirà 0 e `getPage()` continuerà a fallire. Notate che il metodo `decrypt()` decifra solo l'oggetto `PdfFileReader`, non l'effettivo file PDF. Quando il vostro programma termina, il file sul disco fisso rimane cifrato. Il vostro programma dovrà chiamare nuovamente `decrypt()`, quando verrà eseguito la prossima volta.

# Creare PDF

La controparte degli oggetti `PdfFileReader` in `PyPDF2` sono gli oggetti `PdfFileWriter`, che possono **creare nuovi file PDF**. `PyPDF2` però non può scrivere in un PDF testo qualsiasi come invece Python è in grado di fare con i file di puro testo; le capacità di scrittura di `PyPDF2` sono limitate a copiare pagine da altri PDF, ruotare le pagine, sovrapporre le pagine e cifrare i file.

`PyPDF2` non permette di modificare direttamente un PDF; bisogna invece creare un nuovo PDF e poi copiare i contenuti da un documento esistente. Gli esempi in questa sezione seguiranno questa impostazione generale:

- si aprono uno o più PDF esistenti (i PDF sorgente) in oggetti `PdfFileReader`;
- si crea un nuovo oggetto `PdfFileWriter`;
- si copiano pagine dagli oggetti `PdfFileReader` nell'oggetto `PdfFileWriter`;
- infine, si usa l'oggetto `PdfFileWriter` per scrivere il PDF di destinazione.

La creazione di un oggetto `PdfFileWriter` è solo la creazione di un valore che rappresenta un documento PDF in Python: non viene creato il file PDF effettivo – per questa operazione bisogna chiamare il metodo `write()` di `PdfFileWriter`.

Il metodo `write()` prende un normale oggetto `File` che sia stato aperto in modalità scrittura binaria. Si ottiene un oggetto `File` di questo tipo chiamando la funzione `open()` di Python con due argomenti: la stringa contenente quello che si vuole sia il nome di file del PDF e '`wb`' per indicare che il file deve essere aperto in modalità *scrittura binaria*.

Se vi sentite un po' confusi, non preoccupatevi: vedrete come funziona questo procedimento con i prossimi esempi.

## Copiare pagine

Potete usare `PyPDF2` per copiare pagine da un documento PDF a un altro. Questo permette di combinare fra loro più file PDF, di eliminare pagine indesiderate o di riordinare le pagine.

Scaricate `meetingminutes.pdf` e `meetingminutes2.pdf` da <http://nostarch.com/automatestuff/> e salvate i due PDF nella directory di lavoro corrente. Inserite quanto segue nella shell interattiva:

```
>>> import PyPDF2
>>> pdf1File = open('meetingminutes.pdf', 'rb')
>>> pdf2File = open('meetingminutes2.pdf', 'rb')
❶ >>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)
❷ >>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)
❸ >>> pdfWriter = PyPDF2.PdfFileWriter()

❹ >>> for pageNum in range(pdf1Reader.numPages):
    pageObj = pdf1Reader.getPage(pageNum)
    pdfWriter.addPage(pageObj)

❺ >>> for pageNum in range(pdf2Reader.numPages):
    pageObj = pdf2Reader.getPage(pageNum)
    pdfWriter.addPage(pageObj)

❻ >>> pdfOutputFile = open('combinedminutes.pdf', 'wb')
>>> pdfWriter.write(pdfOutputFile)
>>> pdfOutputFile.close()
>>> pdf1File.close()
>>> pdf2File.close()
```

Aprite i due file PDF in modalità lettura binaria e memorizzate i due oggetti File risultanti in pdf1File e pdf2File. Chiamate PyPDF2.PdfFileReader() e gli passate pdf1File per ottenere un oggetto PdfFileReader per *meetingminutes.pdf* ①. Lo chiamate di nuovo e passate pdf2File per avere un oggetto PdfFileReader per *meetingminutes2.pdf* ②. Poi create un nuovo oggetto PdfFileWriter, che rappresenta un documento PDF vuoto ③.

Poi, copiate tutte le pagine dai due PDF sorgente e le aggiungete all'oggetto PdfFileWriter. Ottenere l'oggetto Page chiamando getPage() su un oggetto PdfFileReader ④. Poi passate l'oggetto Page al metodo addPage() del vostro PdfFileWriter ⑤. Questi passi vengono eseguiti prima per pdf1Reader e poi di nuovo per pdf2Reader. Quando avete finito di copiare le pagine, scrivete un nuovo PDF con il nome *combinedminutes.pdf* passando un oggetto File al metodo write() di PdfFileWriter ⑥.

## NOTA

PyPDF2 non può inserire pagine nel mezzo di un oggetto PdfFileWriter; il metodo addPage() è in grado solo di aggiungere pagine alla fine.

Ora avete creato un nuovo file PDF che combina le pagine di *meetingminutes.pdf* e *meetingminutes2.pdf* in un unico documento. Ricordate che l'oggetto File passato a PyPDF2.PdfFileReader() deve essere aperto in modalità lettura binaria, passando 'rb' come secondo argomento a open(). Analogamente, l'oggetto File passato a PyPDF2.PdfFileWriter() deve essere aperto in modalità scrittura binaria con 'wb'.

## Ruotare pagine

Le pagine di un PDF possono essere **ruotate**, per **incrementi di 90 gradi**, con i metodi rotateClockwise() e rotateCounterClockwise(). A questi metodi dovete passare uno degli interi 90, 180 o 270. Inserite quanto segue nella shell interattiva, con il file *meetingminutes.pdf* nella directory di lavoro corrente:

```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❶ >>> page = pdfReader.getPage(0)
❷ >>> page.rotateClockwise(90)
{'/Contents': [IndirectObject(961, 0), IndirectObject(962, 0),
--righe omesse--
]}
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> pdfWriter.addPage(page)
❸ >>> resultPdfFile = open('rotatedPage.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> resultPdfFile.close()
>>> minutesFile.close()
```

Qui usiamo getPage(0) per selezionare la prima pagina del PDF ①, poi chiamiamo rotateClockwise(90) su quella pagina ②. Scriviamo un nuovo PDF con la pagina ruotata e lo salviamo con il nome *rotatedPage.pdf* ③.

Il PDF risultante avrà una sola pagina, ruotata di 90 gradi in senso antiorario, come nella [Figura 13.2](#).

I valori restituiti da `rotateClockwise()` e `rotateCounterClockwise()` contengono molte informazioni che potete ignorare.

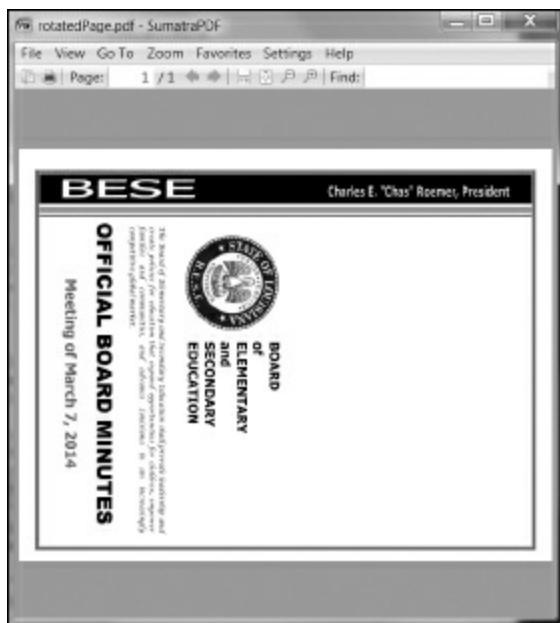


Figura 13.2- Il file rotatedPage.pdf con la pagina ruotata di 90 gradi in senso orario.

## Sovrapporre pagine

PyPDF2 può anche sovrapporre i contenuti di una pagina a quelli di un'altra, cosa utile per aggiungere un logo, un'indicazione oraria o un watermark a una pagina. Con Python è facile aggiungere watermark a più file e solo alle pagine desiderate.

Scaricate `watermark.pdf` da <http://nostarch.com/automatestuff/> e copiate il PDF nella directory di lavoro corrente insieme con `meetingminutes.pdf`. Poi inserite quanto segue nella shell interattiva:

```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
❶ >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❷ >>> minutesFirstPage = pdfReader.getPage(0)
❸ >>> pdfWatermarkReader = PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))
❹ >>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))
❺ >>> pdfWriter = PyPDF2.PdfFileWriter()
❻ >>> pdfWriter.addPage(minutesFirstPage)

❼ >>> for pageNum in range(1, pdfReader.numPages):
    pageObj = pdfReader.getPage(pageNum)
    pdfWriter.addPage(pageObj)
>>> resultPdfFile = open('watermarkedCover.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> minutesFile.close()
>>> resultPdfFile.close()
```

Qui creiamo un oggetto `PdfFileReader` da `meetingminutes.pdf` ❶. Chiamiamo `getPage(0)` per ottenere un oggetto `Page` per la prima pagina e memorizziamo questo oggetto in `minutesFirstPage` ❷. Poi creiamo un oggetto `PdfFileReader` per `watermark.pdf` ❸ e chiamiamo `mergePage()` su `minutesFirstPage` ❹. L'argomento che passiamo a `mergePage()` è un oggetto `Page` per la prima pagina di `watermark.pdf`.

Ora che abbiamo chiamato `mergePage()` su `minutesFirstPage`, quest'ultimo rappresenta la prima pagina con il watermark. Creiamo un oggetto `PdfFileWriter` ❺ e aggiungiamo la prima pagina con il watermark ❻.

Poi cicliamo sul resto delle pagine in *meetingminutes.pdf* e le aggiungiamo all'oggetto PdfFileWriter. Infine, apriamo un nuovo PDF con il nome *watermarkedCover.pdf* e scriviamo i contenuti di PdfFileWriter nel nuovo PDF.

La [Figura 13.3](#) mostra i risultati. Il nostro nuovo PDF, *watermarkedCover.pdf*, ha tutti i contenuti di *meetingminutes.pdf* e la prima pagina presenta il watermark.



**Figura 13.3** - Il PDF originale (a sinistra), con il watermark (al centro) e il PDF risultante (a destra).

## Cifrare PDF

Un oggetto PdfFileWriter può anche cifrare un documento PDF. Inserite quanto segue nella shell interattiva:

```
>>> import PyPDF2
>>> pdfFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFile)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> for pageNum in range(pdfReader.numPages):
    pdfWriter.addPage(pdfReader.getPage(pageNum))

❶ >>> pdfWriter.encrypt('swordfish')
>>> resultPdf = open('encryptedminutes.pdf', 'wb')
>>> pdfWriter.write(resultPdf)
>>> resultPdf.close()
```

Prima di chiamare il metodo `write()` per salvare in un file, chiamate il metodo `encrypt()` e gli passate una stringa con la password **❶**. I PDF possono avere una **password d'utente** (che permette di visualizzare il PDF) e una **password di proprietario** (che permette di fissare le autorizzazioni per la stampa, i commenti, l'estrazione di testo e altre caratteristiche).

La password d'utente e la password di proprietario sono, rispettivamente, il primo e il secondo argomento di `encrypt()`. Se viene passato un solo argomento stringa a `encrypt()`, quella stringa viene usata per entrambe le password.

In questo esempio, abbiamo copiato le pagine di *meetingminutes.pdf* in un oggetto PdfFileWriter. Abbiamo cifrato l'oggetto PdfFileWriter con la password `swordfish`, abbiamo aperto un nuovo PDF con il nome *encryptedminutes.pdf* e abbiamo scritto i contenuti di PdfFileWriter nel nuovo PDF. Ora, se

qualcuno vorrà visualizzare *encryptedminutes.pdf*, dovrà prima inserire questa password. Potete cancellare il file originale *meetingminutes.pdf* non cifrato, dopo aver verificato che la copia sia stata cifrata correttamente.

## Progetto: combinare pagine selezionate da più PDF

Supponiamo che abbiate il noioso compito di fondere varie decine di documenti PDF in un unico file PDF. Ciascun dei documenti ha una copertina come prima pagina, ma non volete che la copertina si ripeta nel risultato finale. Anche se esistono moltissimi programmi gratuiti per combinare PDF, molti possiedono solo la capacità di fondere fra loro file interi. Scriviamo un programma Python che consenta all'utente di selezionare le pagine che vuole compaiano nel PDF combinato.

Ad alto livello, ecco che cosa dovrà fare il programma:

- trovare tutti i file PDF nella directory di lavoro corrente;
- ordinare i nomi dei file, in modo che i PDF vengano combinati in ordine;
- scrivere ciascuna pagina, esclusa la prima, di ciascun PDF nel file finale.

In termini di implementazione, il vostro codice dovrà fare le cose seguenti:

- chiamare `os.listdir()` per trovare tutti i file nella directory di lavoro ed eliminare gli eventuali file che non sono PDF;
- chiamare il metodi di lista `sort()` di Python per mettere in ordine alfabetico i nomi dei file;
- creare un oggetto `PdfFileWriter` per il PDF di output;
- ciclare su ciascun file PDF, creando un corrispondente oggetto `PdfFileReader`;
- ciclare su tutte le pagine (eccetto la prima) in ciascun file PDF;
- aggiungere le pagine al PDF di output;
- scrivere il PDF di output in un file con il nome *allminutes.pdf*.

Per questo progetto, apriate una nuova finestra di file editor e salvate il file con il nome *combinePdfs.py*.

### Passo 1: trovare tutti i file PDF

Come prima cosa, il programma deve prendere una lista di tutti i file con estensione *.pdf* nella directory di lavoro corrente e deve ordinarli alfabeticamente. Scrivete il codice in questo modo:

```

#! python3
# combinePdfs.py - Combina tutti i PDF presenti nella directory corrente
# in un unico PDF.

❶ import PyPDF2, os

# Recupera I nomi di tutti i file PDF.
pdfFiles = []
for filename in os.listdir('.'):
    if filename.endswith('.pdf'):
❷        pdfFiles.append(filename)
❸ pdfFiles.sort(key = str.lower)

❹ pdfWriter = PyPDF2.PdfFileWriter()

# DA FARE: Ciclare su tutti i file PDF.

# DA FARE: Ciclare su tutte le pagine (tranne la prima) e aggiungerle.

# DA FARE: Salvare in un file il PDF risultante.

```

Dopo la riga iniziale e il commento descrittivo su quel che fa il programma, questo codice importa i moduli `os` e `PyPDF2` ❶. La chiamata `os.listdir('')` restituirà una lista con tutti i file nella directory di lavoro corrente. Il codice cicla su questa lista e aggiunge solo i file con estensione `.pdf` a `pdfFiles` ❷. Poi la lista viene ordinata alfabeticamente con l'argomento per parola chiave `key = str.lower` passato a `sort()` ❸. Viene creato un oggetto `PdfFileWriter` che contenga le pagine PDF combinate ❹. Infine, alcuni commenti delineano il resto del programma.

## Passo 2: aprire i singoli PDF

Ora il programma deve leggere ciascuno dei file in `pdfFiles`. Aggiungete quanto segue al vostro programma:

```

#! python3
# combinePdfs.py - Combina tutti i PDF presenti nella directory corrente
# in un unico PDF.

import PyPDF2, os

# Prende tutti i nomi dei file PDF.
pdfFiles = []
--righe omesse--

# Cicla su tutti i file PDF.
for filename in pdfFiles:
    pdfFileObj = open(filename, 'rb')
    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
    # DA FARE: Ciclare su tutte le pagine (tranne la prima) e aggiungerle.

# DA FARE: Salvare in un file il PDF risultante.

```

Per ciascun PDF, il ciclo apre un nome di file in modalità lettura binaria chiamando `open()` con `'rb'` come secondo argomento. La chiamata `open()` restituisce un oggetto `File` che viene passato a `PyPDF2.PdfFileReader()` per creare un oggetto `PdfFileReader` per quel file PDF.

## Passo 3: aggiungere le singole pagine

Per ciascun PDF, vogliamo ciclare su tutte le pagine tranne la prima. Aggiungete questo codice al vostro programma:

```
#! python3
# combinePdfs.py - Combina tutti i PDF presenti nella directory corrente
# in un unico PDF.

import PyPDF2, os

--righe omesse--

# Cicla su tutti i file PDF.
for filename in pdfFiles:

--righe omesse--

❶ # Cicla su tutte le pagine (tranne la prima) e le aggiunge.
    for pageNum in range(1, pdfReader.numPages):
        pageObj = pdfReader.getPage(pageNum)
        pdfWriter.addPage(pageObj)

# DA FARE: Salvare in un file il PDF risultante.
```

Il codice nel ciclo for copia ciascun oggetto Page individualmente nell'oggetto PdfFileWriter. Ricordate: volete saltare la prima pagina. Poiché PyPDF2 considera 0 la prima pagina, il vostro ciclo deve iniziare da 1 ❶ e poi proseguire fino all'intero (escluso) in pdfReader.numPages.

## Passo 4: salvare il risultato

Quando i cicli for annidati hanno svolto il loro lavoro, la variabile pdfWriter conterrà un oggetto PdfFileWriter con le pagine di tutti i PDF combinati. L'ultimo passo è scrivere questi contenuti in un file sul disco fisso.

Aggiungete questo codice al vostro programma:

```

#! python3
# combinePdfs.py - Combina tutti i PDF presenti nella directory corrente
# in un unico PDF.

import PyPDF2, os

--righe omesse--

# Cicla su tutti i file PDF.
for filename in pdfFiles:

--righe omesse--

# Cicla su tutte le pagine (tranne la prima) e le aggiunge.
for pageNum in range(1, pdfReader.numPages):

--righe omesse--

# Salva in un file il PDF risultante.
pdfOutput = open('allminutes.pdf', 'wb')
pdfWriter.write(pdfOutput)
pdfOutput.close()

```

Passando 'wb' a `open()` si apre il file PDF di *output*, *allminutes.pdf*, in modalità scrittura binaria. Poi, passando l'oggetto File risultante al metodo `write()` si crea il file PDF effettivo. Una chiamata al metodo `close()` conclude il programma.

## Idee per programmi simili

Poter creare PDF dalle pagine di altri PDF vi permetterà di scrivere programmi che facciano cose come queste:

- eliminare pagine specifiche dai PDF;
- riordinare le pagine in un PDF;
- creare un PDF esclusivamente con le pagine che contengono un testo specifico, identificato da `extractText()`.

## Documenti Word

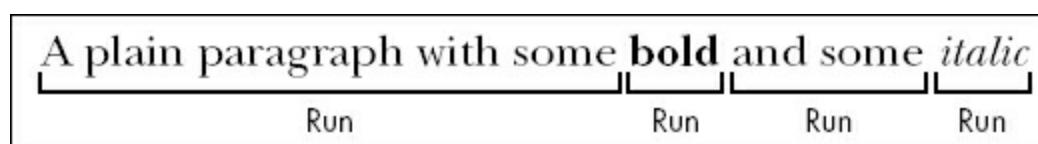
Python può creare e modificare documenti Word, che hanno l'estensione di file *.docx*, con il modulo `python-docx`. Potete installare questo modulo eseguendo `pip install python-docx` dalla riga di comando. (L'[Appendice A](#) presenta le istruzioni dettagliate per l'installazione di moduli di terze parti.)

### NOTA

Quando usate `pip` per installare Python-Docx fate attenzione a installare `python-docx`, non `docx`. Il nome `docx` indica un modulo diverso, di cui in questo libro non parleremo. Quando però dovrete importare il modulo `python-docx`, dovrete eseguire `import docx`, non `import python-docx`.

alternative e libere per Windows, OS X e Linux, che possono essere utilizzate per aprire file .docx. Potete scaricarli da <https://www.libreoffice.org> e <http://openoffice.org>, rispettivamente. La documentazione completa di Python-Docx è disponibile all'indirizzo <https://python-docx.readthedocs.org/>. Esiste anche una versione di Word per OS X, ma questo capitolo si concentrerà su Word per Windows.

Rispetto al puro testo, i file .docx hanno una gran quantità di struttura. Questa struttura è rappresentata da tre diversi tipi di dati in Python-Docx. Al livello più alto, un oggetto Document rappresenta il documento nella sua totalità. L'oggetto Document contiene una lista di oggetti Paragraph per i paragrafi nel documento. (Inizia un nuovo paragrafo ogni volta che l'utente preme Invio mentre scrive in un documento Word.) Ciascuno di questi oggetti Paragraph contiene una lista di uno o più oggetti Run. Il paragrafo, costituito da un'unica frase, nella [figura 13.4](#), contiene quattro Run.



**Figura 13.4** - Gli oggetti Run identificati in un oggetto Paragraph.

Il testo in un documento Word è molto più di una semplice stringa: ha associate anche informazioni su tipi di caratteri, dimensioni, colori e altre informazioni di stile. In Word uno stile è un insieme di questi attributi. Un oggetto Run è una porzione contigua di testo con lo stesso stile: è necessario un nuovo oggetto Run a ogni cambio di stile.

## Leggere documenti Word

Proviamo a fare qualche esperimento con il modulo python-docx. Scaricate il file demo.docx da <http://nostarch.com/automatestuff/> e salvatelo nella directory di lavoro corrente. Poi inserite quanto segue nella shell interattiva:

```
>>> import docx
❶ >>> doc = docx.Document('demo.docx')
❷ >>> len(doc.paragraphs)
7
❸ >>> doc.paragraphs[0].text
'Document Title'
❹ >>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
❺ >>> len(doc.paragraphs[1].runs)
4
❻ >>> doc.paragraphs[1].runs[0].text
'A plain paragraph with some '
❼ >>> doc.paragraphs[1].runs[1].text
'bold'
❽ >>> doc.paragraphs[1].runs[2].text
' and some '
❾ >>> doc.paragraphs[1].runs[3].text
'italic'
```

In ❶, apriamo un file .docx in Python, chiamiamo docx.Document(), passando il nome di file *demo.docx*. Questo restituisce un oggetto Document, che ha un attributo paragraphs, una lista di oggetti Paragraph.

Quando chiamiamo `len()` su `doc.paragraphs`, ci restituisce 7, il che ci dice che in questo documento vi sono sette oggetti `Paragraph` ②. Ciascuno di questi oggetti `Paragraph` ha un attributo `text` che contiene una stringa del testo di quel paragrafo (senza le informazioni sullo stile). Qui il primo attributo `text` contiene 'DocumentTitle' ③, il secondo contiene 'A plain paragraph with some bold and some italic' ④. La chiamata di `len()` su questo oggetto `Paragraph` ci dice che vi sono quattro oggetti `Run` ⑤. Il primo oggetto `Run` contiene 'A plain paragraph with some' ⑥. Poi il testo passa a uno stile grassetto, perciò 'bold' inizia un nuovo oggetto `Run` ⑦. Il testo ritorna subito dopo a uno stile tondo normale, il che produce un terzo oggetto `Run`, ' and some' ⑧. Infine, il quarto e ultimo oggetto `Run` contiene 'italic' con uno stile corsivo ⑨.

Con Python-Docx, i vostri programmi Python ora saranno in grado di leggere il testo da un file `.docx` e di usarlo come qualsiasi altro valore stringa.

## Ottenere il testo completo da un file `.docx`

Se vi interessa solo il testo del documento Word, e non le informazioni sugli stili, potete utilizzare la funzione `getText()`, che accetta il nome di un file `.docx` e restituisce un singolo valore stringa con il suo testo. Aprite una nuova finestra di file editor e inserite il codice seguente, poi salvatelo come `readDocx.py`.

```
#! python3

import docx

def getText(filename):
    doc = docx.Document(filename)
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
return '\n'.join(fullText)
```

La funzione `getText()` apre il documento Word, cicla su tutti gli oggetti `Paragraph` nella lista `paragraphs` e poi ne accoda il testo alla lista in `fullText`. Dopo il ciclo, le stringhe in `fullText` vengono unite fra loro con caratteri di a capo (`newline`).

Il programma `readDocx.py` può essere importato come qualsiasi altro modulo. Ora se vi serve semplicemente il puro testo di un documento Word, potete inserire quanto segue:

```
>>> import readDocx
>>> print(readDocx.getText('demo.docx'))
Document Title
A plain paragraph with some bold and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list
```

Potete anche regolare `getText()` in modo da modificare la stringa prima che venga restituita. Per esempio, per rientrare tutti i paragrafi, sostituite la chiamata `append()` in `readDocx.py` con questo enunciato:

```
fullText.append(' ' + para.text)
```

Per aggiungere uno spazio doppio fra paragrafi, modificate la chiamata `join()` così:

```
return '\n\n'.join(fullText)
```

Come potete vedere, bastano poche righe di codice per scrivere funzioni che leggano un file `.docx` e restituiscano una stringa con i suoi contenuti, nella forma che preferite.

## Stili di paragrafo e oggetti Run

In Word per Windows, si possono vedere gli stili premendo la combinazione di tasti `Ctrl-Alt-Maiusc-S`: il pannello con gli stili si presenta come nella [figura 13.5](#). In OS X, si può vedere il pannello *Stili* selezionando da menu **Visualizza > Stili**.



**Figura 13.5** - In Windows, il pannello Stili si può visualizzare premendo `Ctrl-Alt-Maiusc-S`.

Word e altri elaboratori di testi usano gli stili per garantire la coerenza della presentazione visiva di componenti testuali simili, e per rendere più facile la manutenzione. Per esempio, poniamo che vogliate impostare i paragrafi di testo in Times New Roman, 11 punti, allineati a sinistra. Potete creare uno stile con queste impostazioni e assegnarlo a tutti i paragrafi di testo. Poi, se in seguito volete modificare l'aspetto di tutti i paragrafi di testo nel documento, basta che modifichiate lo stile, e tutti quei paragrafi verranno aggiornati automaticamente.

Per i documenti di Word, esistono tre tipi di stili: gli *stili di paragrafo* possono essere applicati agli oggetti *Paragraph*, gli *stili di carattere* possono essere applicati agli oggetti *Run*, e gli *stili collegati* possono essere applicati a entrambi i tipi di oggetti. Potete attribuire uno stile sia agli oggetti *Paragraph* sia agli oggetti *Run* impostando il loro attributo `style` a una stringa. Questa stringa deve essere il nome

di uno stile. Se `style` è impostato a `None`, allora non ci sarà alcuno stile associato all'oggetto `Paragraph` o `Run`.

I valori stringa per gli stili predefiniti di Word sono questi:

'Normal'	'Heading5'	'ListBullet'	'ListParagraph'
'BodyText'	'Heading6'	'ListBullet2'	'MacroText'
'BodyText2'	'Heading7'	'ListBullet3'	'NoSpacing'
'BodyText3'	'Heading8'	'ListContinue'	'Quote'
'Caption'	'Heading9'	'ListContinue2'	'Subtitle'
'Heading1'	'IntenseQuote'	'ListContinue3'	'TOCHeading'
'Heading2'	'List'	'ListNumber'	'Title'
'Heading3'	'List2'	'ListNumber2'	
'Heading4'	'List3'	'ListNumber3'	

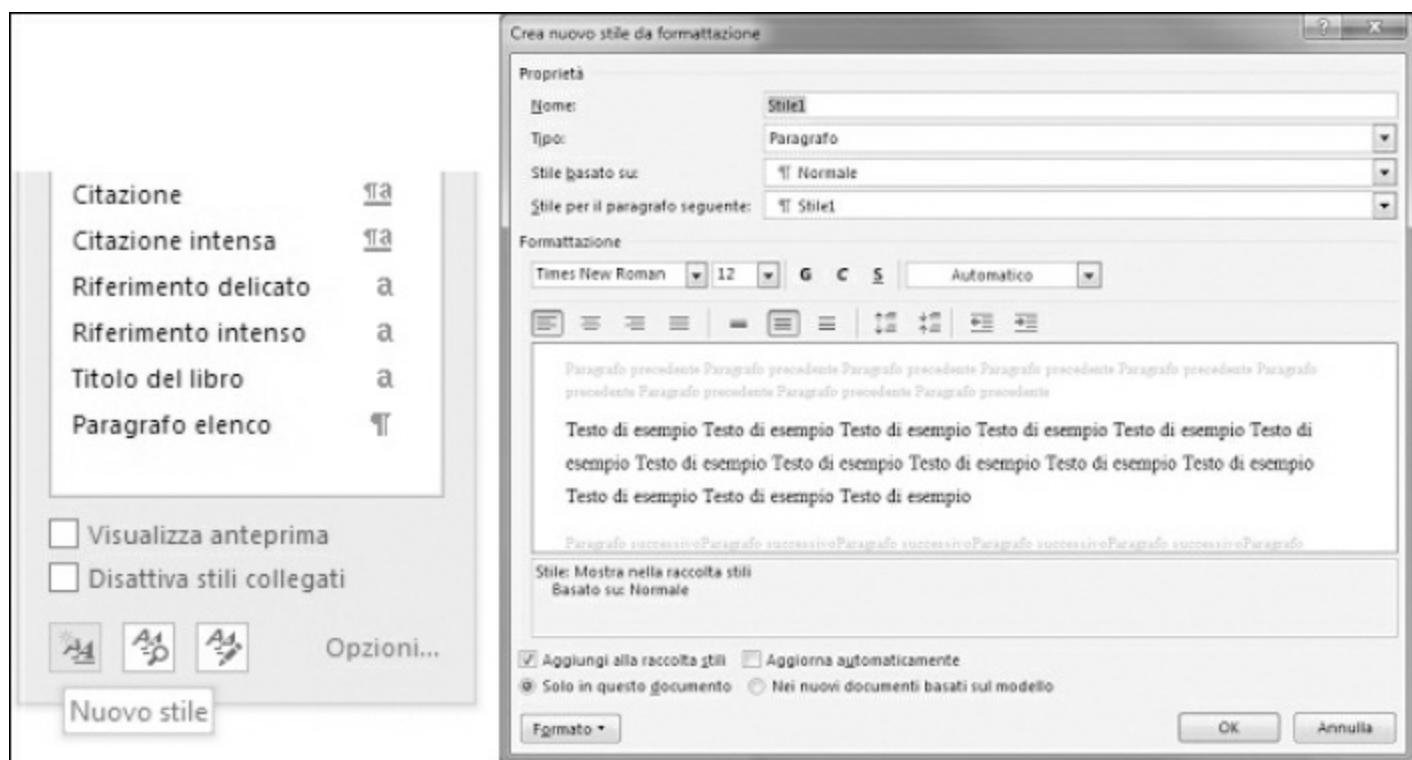
Notate che gli stili predefiniti sono memorizzati da Word in un file *WordprocessingML* con il loro nome inglese, per esempio *Heading 1*, anche se l'utente vede quello stile, nell'interfaccia di Word, localizzato, per esempio come *Titolo 1*. Il modulo `python-docx` lavora sul file *WordprocessingML*, perciò usa sempre i nomi inglesi degli stili. Gli stili definiti dall'utente, invece, non sono conservati in quel file e perciò vi si accede con lo stesso nome che compare nell'interfaccia utente. Per la corrispondenza fra i nomi degli stili in inglese e quelli in altre lingue, potete consultare [http://www.thedoctools.com/index.php?show=met\\_create\\_style\\_name\\_list](http://www.thedoctools.com/index.php?show=met_create_style_name_list).

Quando impostate l'attributo `style`, non usate spazi nel nome dello stile. Per esempio, il nome dello stile può essere *Subtle Emphasis*, ma dovete impostare l'attributo `style` al valore stringa '`SubtleEmphasis`' e non '`Subtle Emphasis`'. L'inclusione degli spazi farebbe sì che Word interpretasse erroneamente il nome dello stile e non lo applicasse.

Quando si usa uno stile collegato per un oggetto `Run`, bisogna aggiungere '`Char`' alla fine del suo nome. Per esempio, per impostare lo stile collegato *Quote* per un oggetto paragrafo, si userà `paragraphObj.style = 'Quote'`, ma per un oggetto `Run` si deve usare `runObj.style = 'QuoteChar'`.

## Creare documenti di Word con stili non predefiniti

Se volete creare documenti Word che usano stili diversi da quelli predefiniti, dovete aprire Word con un documento vuoto e creare gli stili facendo clic sul pulsante *Nuovo stile*, che si trova nella parte inferiore del riquadro *Stili* (la [Figura 13.6](#) lo evidenzia, per Word per Windows).



**Figura 13.6** - Il pulsante Nuovo stile (a sinistra) e la finestra di dialogo Crea nuovo stile da formattazione (a destra).

Il pulsante apre la finestra di dialogo *Crea nuovo stile da formattazione*, in cui si può inserire il nuovo stile. Poi tornate alla shell interattiva e aprirete il documento Word vuoto con docx.Document(), in modo da poterlo utilizzare come base per il vostro documento Word. Il nome che attribuirete allo stile a quel punto sarà disponibile per l'uso con Python-Docx.

## Attributi Run

Lo stile per gli oggetti Run può essere perfezionato utilizzando gli attributi text. Ciascun attributo può essere impostato a uno fra tre valori: True (l'attributo è sempre abilitato, non importa quali altri stili siano applicati a quell'elemento), False (l'attributo è sempre disabilitato), oppure None (l'elemento riceve di default lo stile a cui è preimpostato).

La [Tabella 13.1](#) elenca gli attributi text che possono essere impostati per gli oggetti Run.

**Tabella 13.1** - Gli attributi text per oggetti Run.

Attributo	Descrizione
bold	Il testo è in grassetto.
italic	Il testo è in corsivo.
underline	Il testo è sottolineato.
strike	Il testo è barrato.
double_strike	Il testo è doppiamente barrato.
all_caps	Il testo è in tutte maiuscole.
small_caps	Il testo è in tutte maiuscole, con le minuscole di due punti più piccole.

shadow	Il testo è ombreggiato.
outline	Il testo è solo un contorno.
rtl	Il testo è scritto da destra a sinistra.
imprint	Il testo appare scavato nella pagina.
emboss	Il testo appare in rilievo sulla pagina.

Per esempio, per modificare gli stili di demo.docx, inserite quanto segue nella shell interattiva:

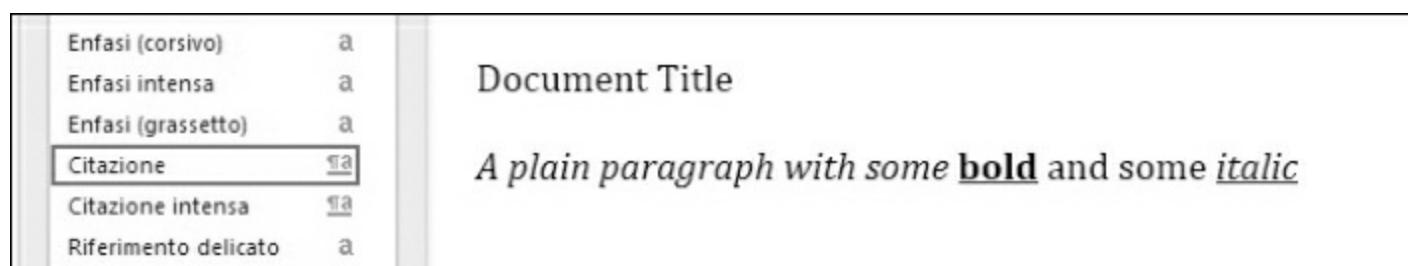
```
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style
'Title'
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc.
paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'QuoteChar'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

Qui usiamo gli attributi `text` e `style` per vedere facilmente che cosa si trova nei paragrafi del nostro documento.

Si può vedere che è semplice dividere un paragrafo in run e accedere a ciascun run individualmente. Così possiamo ottenere il primo, il secondo e il quarto run nel secondo paragrafo, possiamo attribuire uno stile a ciascuno, e salvare il risultato in un nuovo documento.

Le parole *Document Title* all'inizio di *restyled.docx* avranno lo stile *Normal* (Normale) anziché lo stile *Title* (*Titolo*) l'oggetto *Run* per il testo *A plain paragraph with some* avrà lo stile *QuoteChar* (*Citazione*) e i due oggetti *Run* per le parole *bold* e *italic* avranno i loro attributi `underline` (di sottolineatura) impostati a `True`.

La [Figura 13.7](#) mostra come appaiono gli stili di paragrafi e run in *restyled.docx*.



**Figura 13.7** - Il file restyled.docx.

Potete trovare una documentazione più completa sull'uso degli stili in Python-Docx all'indirizzo <http://python-docx.readthedocs.org/en/latest/user/styles.html>.

# Scrivere documenti Word

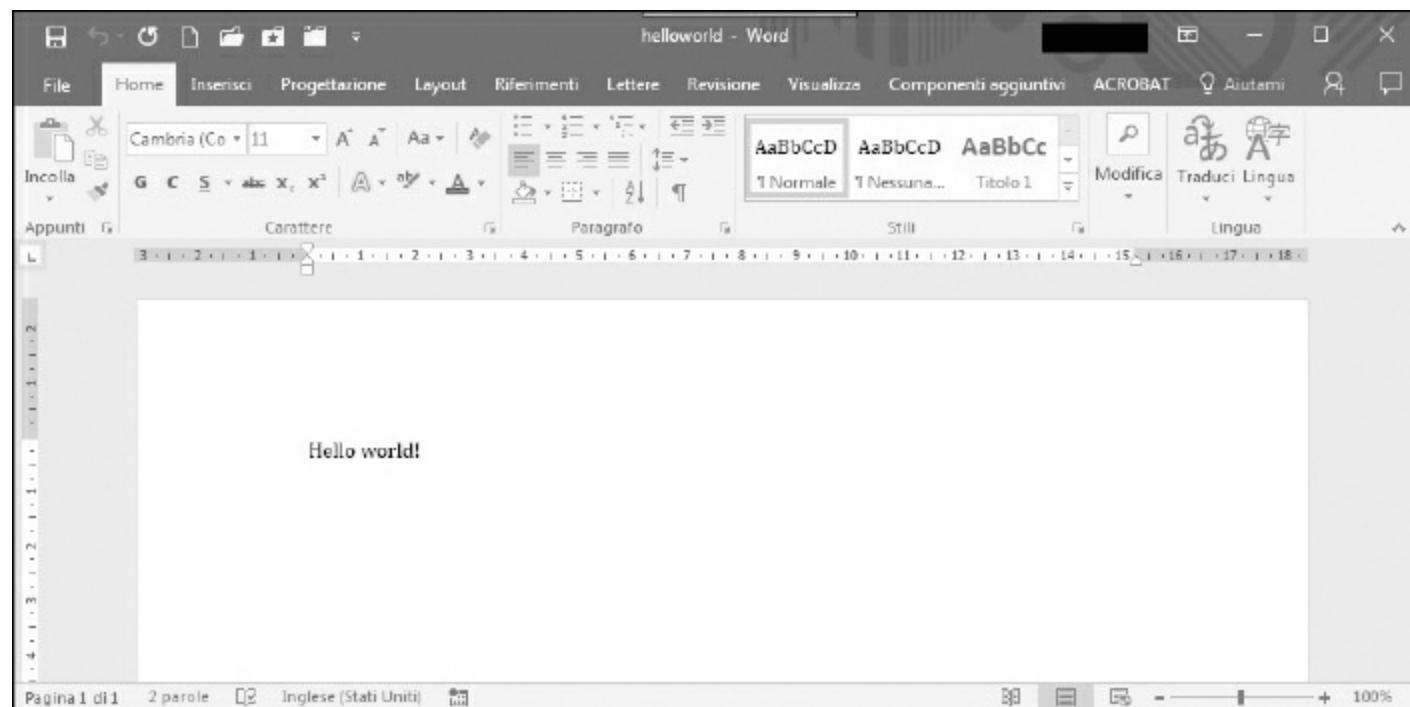
Inserite quanto segue nella shell interattiva:

```
>>> import docx  
>>> doc = docx.Document()  
>>> doc.add_paragraph('Hello world!')  
<docx.text.Paragraph object at 0x000000003B56F60>  
>>> doc.save('helloworld.docx')
```

Per creare un vostro file *.docx*, chiamate `docx.Document()`, che vi restituirà un nuovo oggetto `Document` vuoto di Word. Il metodo `add_paragraph()` del documento aggiunge un nuovo paragrafo di testo al documento e restituisce un riferimento all'oggetto `Paragraph` che è stato aggiunto.

Quando avete finito di aggiungere testo, passate una stringa con il nome di file al metodo `save()` del documento per salvare l'oggetto `Document` in un file.

Questo creerà un file con il nome *helloworld.docx* nella directory di lavoro corrente, che, quando viene aperto, si presenta come nella [Figura 13.8](#).



**Figura 13.8** - Il documento word creato utilizzando `add_paragraph('Hello world!')`.

Potete aggiungere paragrafi chiamando di nuovo il metodo `add_paragraph()` con il testo del nuovo paragrafo. Altrimenti, per aggiungere testo alla fine di un paragrafo già esistente, potete chiamare il metodo `add_run()` del paragrafo, passandogli come argomento una stringa.

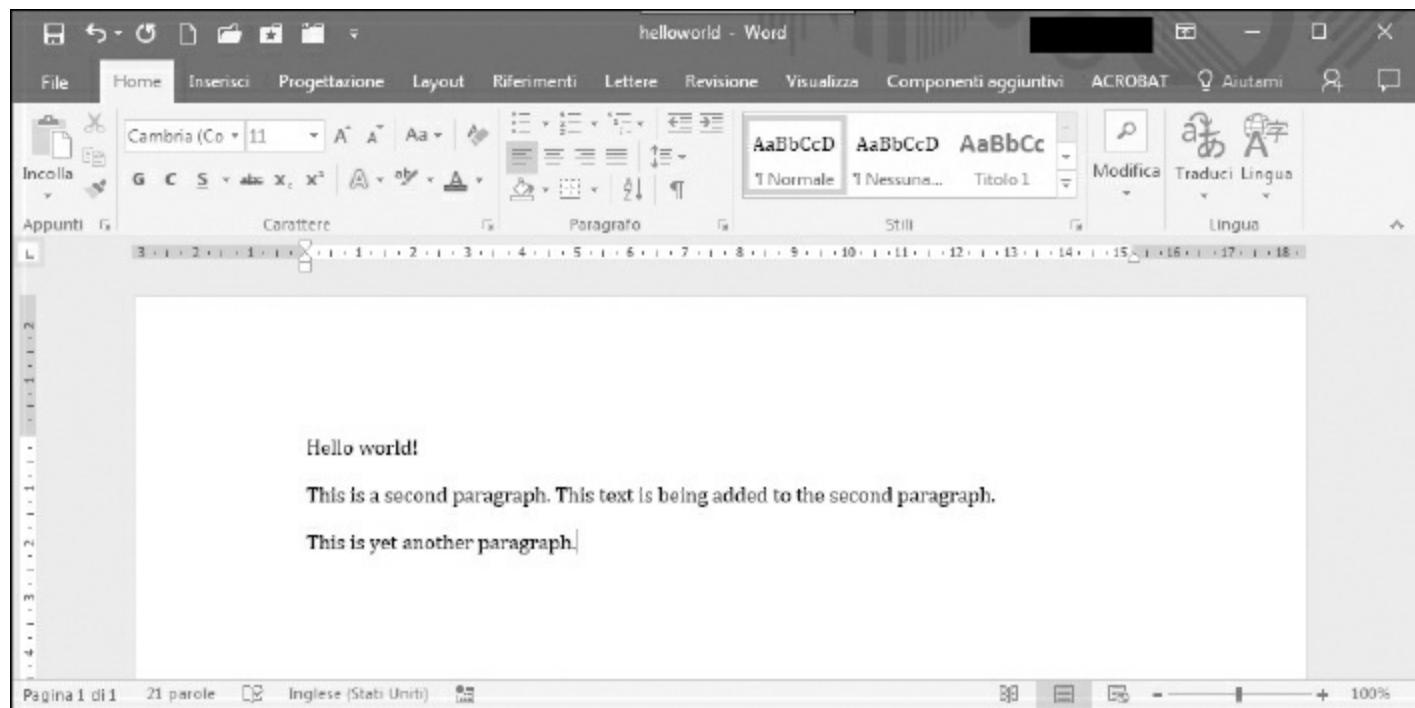
Inserite quanto segue nella shell interattiva:

```
>>> import docx  
>>> doc = docx.Document()  
>>> doc.add_paragraph('Hello world!')  
<docx.text.Paragraph object at 0x00000000366AD30>  
>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')  
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')  
>>> paraObj1.add_run(' This text is being added to the second paragraph.')
```

```
<docx.text.Run object at 0x00000000003A2C860>
```

```
>>> doc.save('multipleParagraphs.docx')
```

Il documento risultante si presenterà come nella [Figura 13.9](#).



**Figura 13.9** - Il documento con vari oggetti Paragraph e Run aggiunti.

Notate che il testo *This text is being added to the second paragraph* è stato aggiunto all'oggetto Paragraph in paraObj1, che era il secondo paragrafo aggiunto a doc. Le funzioni add\_paragraph() e add\_run() restituiscono oggetti Paragraph e Run, rispettivamente, per farvi risparmiare il tempo necessario per estrarli con un passo distinto.

Ricordate che, fino alla versione 0.5.3 di Python-Docx, i nuovi oggetti Paragraph possono essere aggiunti solo alla fine del documento e i nuovi oggetti Run possono essere aggiunti solo alla fine di un oggetto Paragraph.

Il metodo save() può essere chiamato nuovamente per salvare i cambiamenti ulteriori che sono stati apportati.

Sia add\_paragraph() che add\_run() accettano un secondo argomento facoltativo, che è una stringa dello stile per l'oggetto Paragraph o Run. Per esempio:

```
>>> doc.add_paragraph('Hello world!', 'Title')
```

Questa riga aggiunge un paragrafo con il testo *Hello world!* nello stile *Title*.

## Aggiungere titoli

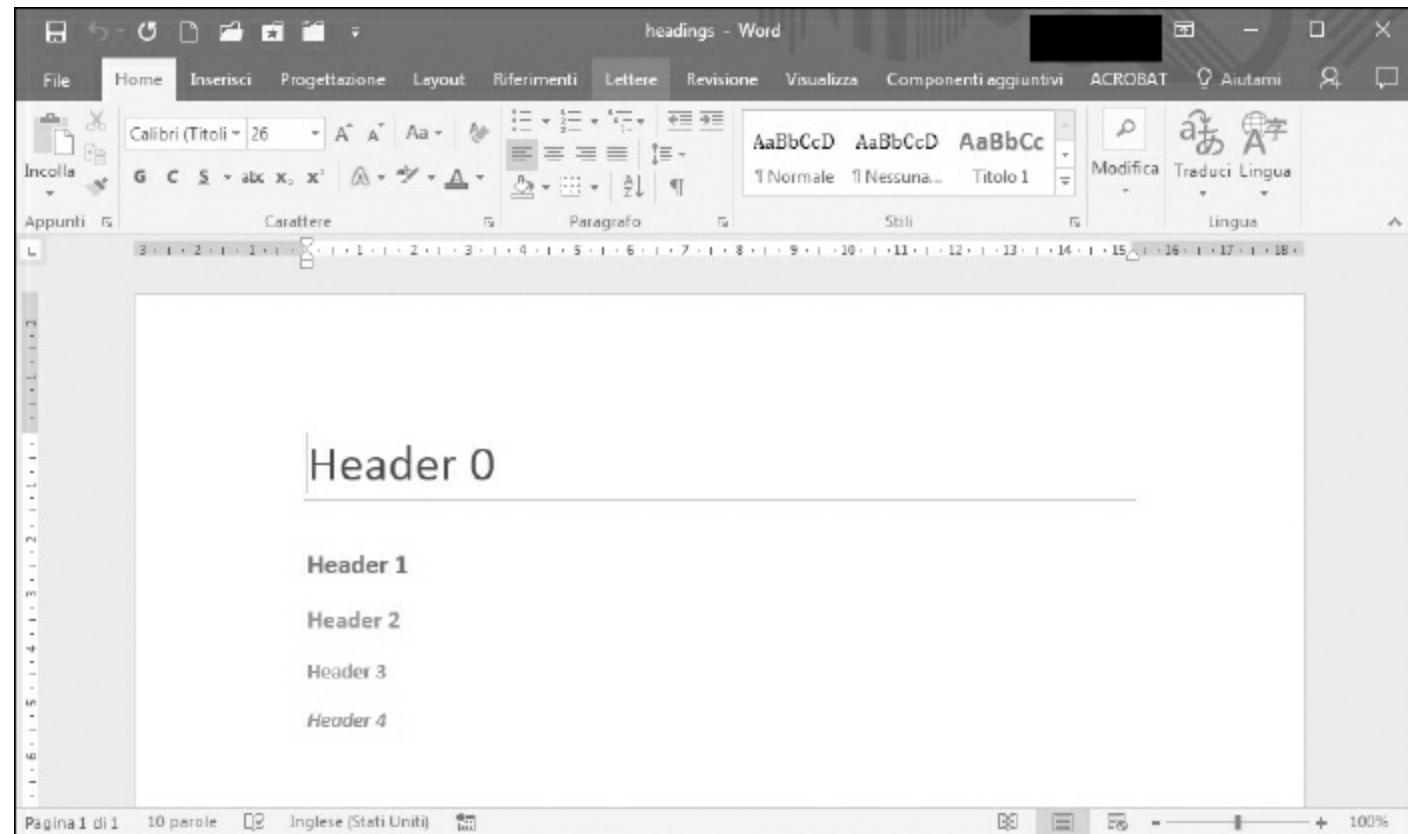
Se si chiama add\_heading(), si aggiunge un paragrafo con uno degli stili di titolo (*heading*). Inserite quanto segue nella shell interattiva:

```
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)
<docx.text.Paragraph object at 0x000000000036CB3C8>
```

```
>>> doc.add_heading('Header 1', 1)
<docx.text.Paragraph object at 0x00000000036CB630>
>>> doc.add_heading('Header 2', 2)
<docx.text.Paragraph object at 0x00000000036CB828>
>>> doc.add_heading('Header 3', 3)
<docx.text.Paragraph object at 0x00000000036CB2E8>
>>> doc.add_heading('Header 4', 4)
<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.save('headings.docx')
```

Gli argomenti di `add_heading()` sono una stringa con il testo del titolo e un intero compreso fra 0 e 4. L'intero 0 attribuisce al testo lo stile *Title* (*Titolo*) utilizzato di norma per il titolo del documento nel suo complesso; gli interi da 1 a 4 sono per i vari livelli di titolo (*Titolo 1*, *Titolo 2* ecc. nell'edizione italiana di Word). La funzione `add_heading()` restituisce un oggetto `Paragraph` per risparmiarvi di dover prevedere un passo in più per estrarlo dall'oggetto `Document`.

Il file *headings.docx* risultante si presenterà come nella [Figura 13.10](#).



## Aggiungere interruzioni di riga e di pagina

Per inserire un'interruzione di riga (anziché iniziare un nuovo paragrafo) si può chiamare il metodo `add_break()` sull'oggetto `Run` dopo il quale si vuole l'interruzione.

Se volete inserire, anziché un'interruzione di riga, un'interruzione di pagina, dovete passare il valore `docx.text.WD_BREAK.PAGE` come unico argomento di `add_break()`, come si vede in questo esempio:

```
>>> doc = docx.Document()
>>> doc.add_paragraph('Questo sta sulla prima pagina!')
<docx.text.Paragraph object at 0x0000000003785518>
❶ >>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
>>> doc.add_paragraph(' Questo sta sulla seconda pagina! ')
<docx.text.Paragraph object at 0x00000000037855F8>
>>> doc.save('twoPage.docx')
```

Questo crea un documento Word di due pagine con *Questo sta sulla prima pagina!* sulla prima pagina e *Questo sta sulla seconda pagina!* sulla seconda. Anche se c'è ancora molto spazio sulla prima pagina dopo quel breve testo, abbiamo impostato che il paragrafo successivo iniziasse su una nuova pagina inserendo una interruzione di pagina dopo il primo run del primo paragrafo ❶.

## Aggiungere immagini

Gli oggetti Document hanno un metodo `add_picture()` che consente di aggiungere un'immagine alla fine del documento. Poniamo che abbiate un file *zophie.png* nella directory di lavoro corrente: potete aggiungere *zophie.png* alla fine del vostro documento con una larghezza di 1 pollice e un'altezza di 4 centimetri (Word può utilizzare sia le unità imperiali sia quelle metriche), scrivendo:

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1), height=docx.shared.Cm(4))
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

Il primo argomento è una stringa che contiene il nome di file dell'immagine. Gli argomenti per parola chiave, facoltativi, `width` e `height`, impostano rispettivamente larghezza e altezza dell'immagine nel documento. Se sono omessi, larghezza e altezza saranno per default le dimensioni normali dell'immagine.

Probabilmente preferirete specificare altezza e larghezza di un'immagine in unità che vi sono ben note, per esempio centimetri o pollici, perciò potete usare le funzioni `docx.shared.Cm()` o `docx.shared.Inches()` quando specificate gli argomenti per parola chiave `width` e `height`.

## Riepilogo

Le **informazioni testuali** non riguardano solo i file di puro testo; in effetti, è molto probabile che vi capitî più spesso di avere a che fare con documenti **PDF** e **Word**. Potete usare il modulo PyPDF2 per leggere e scrivere documenti PDF. Purtroppo, la **lettura da documenti PDF** è possibile che non sempre abbia come risultato una traduzione perfetta in una stringa perché il formato di file PDF è complesso, al punto che alcuni PDF potrebbero risultare non leggibili. In questi casi non c'è molto da fare, a meno che futuri aggiornamenti di PyPDF2 supportino ulteriori caratteristiche del formato PDF.

I **documenti Word** sono più affidabili, e li si può leggere con il modulo `python-docx`. Potete **manipolare testo** nei documenti Word attraverso gli oggetti `Paragraph` e `Run`. A questi oggetti possono anche essere attribuiti degli **stili**, che possono essere stili predefiniti o stili già presenti nel documento. Potete **aggiungere** nuovi paragrafi, titoli, interruzioni e immagini al documento, ma solo alla fine.

Molti dei **limiti** che si incontrano nel lavorare con documenti PDF e Word derivano dal fatto che questi tipi di formati sono pensati per una visualizzazione elegante per gli esseri umani, non per la facilità di analisi da parte del software. Nel prossimo capitolo esamineremo due altri formati comuni

per la memorizzazione delle informazioni: i file JSON e CSV. Questi formati sono stati pensati per l'uso da parte di computer e vedremo che Python può manipolarli molto più facilmente.

## Domande di ripasso

1. Alla funzione PyPDF2.PdfFileReader() non viene passato il nome di file di un PDF. Che cosa viene passato invece?
2. In quali modalità devono essere aperti gli oggetti File per PdfFileReader() e PdfFileWriter()?
3. Come si ottiene un oggetto Page per la pagina 5 di un oggetto PdfFileReader?
4. Quale variabile PdfFileReader memorizza il numero delle pagine nei documenti PDF?
5. Se il PDF di un oggetto PdfFileReader è cifrato con la password swordfish, che cosa dovete fare prima di poterne ottenere degli oggetti Page?
6. Quali metodi si usano per ruotare una pagina?
7. Quale metodo restituisce un oggetto Document per un file che si chiama *demo.docx*?
8. Qual è la differenza fra un oggetto Paragraph e un oggetto Run?
9. Come si ottiene una lista di oggetti Paragraph per un oggetto Document che è memorizzato in una variabile che si chiama doc?
10. Che tipo di oggetto possiede variabili bold, underline, italic, strike e outline?
11. Qual è la differenza fra le impostazioni True, False e None della variabile bold?
12. Come si crea un oggetto Document per un nuovo documento Word?
13. Come si aggiunge un paragrafo con il testo 'Hello there!' in un oggetto Document memorizzato in una variabile che si chiama doc?
14. Quali interi rappresentano i livelli di titolo disponibili nei documenti Word?

## Un po' di pratica

Per esercitarvi, scrivete dei programmi che svolgano le attività seguenti.

## PDF Paranoia

Utilizzando la funzione os.walk() vista nel [Capitolo 9](#), realizzate uno script che prenda ogni PDF che si trova in una cartella (e nelle sue sottocartelle) e cifri il file utilizzando una password indicata sulla riga di comando. Salvate i PDF cifrati con un suffisso *\_encrypted.pdf* aggiunto al nome di file originale. Prima di cancellare il file originale, il programma deve provare a leggere e decifrare il file, per darvi la garanzia che sia stato cifrato correttamente.

Poi, scrivete un programma che trovi tutti i PDF cifrati in una cartella (e nelle sue sottocartelle) e crei una copia in chiaro del PDF utilizzando una password che viene fornita. Se la password è errata, il programma deve stampare un messaggio per l'utente e procedere con il PDF successivo.

## Inviti personalizzati come documenti Word

Supponiamo che abbiate un file di testo con i nomi di vari ospiti. Questo file *guests.txt* ha un nome per riga:

Prof. Plan  
Miss Scarlet  
Col. Mustard

Scrivete un programma che generi un documento Word contenente un invito personalizzato, simile a quello della [Figura 13.11](#).



**Figura 13.11** - Il documento Word generato dallo script personalizzato per gli inviti.

Poiché Python-Docx può usare solo gli stili già esistenti nel documento Word, dovrete prima aggiungere questi stili a un file Word vuoto e successivamente aprire quel file con Python-Docx. Nel documento Word risultante deve esserci un invito solo per pagina, perciò chiamate `add_break()` per inserire un'interruzione di pagina dopo l'ultimo paragrafo di ciascun invito. In questo modo, dovete aprire un solo documento Word per stampare i vostri inviti in una volta sola.

Potete scaricare un file `guests.txt` di esempio da <http://nostarch.com/automatestuff/>.

## Un programma per individuare a forza bruta la password di un PDF

Supponiamo che abbiate un file PDF cifrato di cui avete dimenticato la password, ma ricordate che si trattava di una singola parola inglese. Cercare di indovinare la password dimenticata potrebbe essere lungo e noioso; potete invece scrivere un programma che decifri il PDF provando ogni possibile parola inglese, fino a che non trova quella che funziona. Questo è un **attacco a forza bruta**. Scaricate il file di testo `dictionary.txt` da <http://nostarch.com/automatestuff/>. Questo file di dizionario contiene oltre 44.000 parole inglesi, una parola per riga.

Utilizzando le abilità di lettura dei file apprese nel [Capitolo 8](#), create una lista di stringhe leggendo il file; poi ciclate su ciascuna parola nella lista, passandola al metodo `decrypt()`. Se il metodo restituisce l'intero 0, la password era sbagliata e il programma deve passare alla parola successiva. Se `decrypt()` restituisce 1, il programma deve uscire dal ciclo e stampare la password individuata. Dovete provare sia la versione minuscola sia quella maiuscola di ciascuna parola. (Sul mio laptop, l'esame di tutte le 88.000 parole, fra maiuscole e minuscole, del dizionario richiede solo un paio di minuti: è questo il

motivo per cui non dovete mai usare una semplice parola inglese come password.)

# Lavorare con file CSV e dati JSON

Nel **Capitolo 13**, avete visto come estrarre testo da documenti PDF e Word, file in un formato binario che richiedevano moduli speciali di Python per poter accedere ai loro dati. I file **CSV** e **JSON**, invece, sono **file di puro testo** e li si può esaminare in un editor di testo, come il file editor di IDLE. Python però dispone anche di **moduli speciali** `csv` e `json`, i quali mettono a disposizioni **funzioni** che facilitano il lavoro con questi formati di file.

CSV sta per **comma-separated values** (valori separati da virgole) e i file CSV sono **fogli di calcolo** semplificati salvati come file di **puro testo**. Il modulo `csv` di Python rende facile analizzare questo tipo di file.

JSON è un formato che memorizza le informazioni sotto forma di **codice sorgente JavaScript** in file di puro testo. (JSON è l'acronimo di **JavaScript Object Notation**.) Non è necessario conoscere il linguaggio di programmazione JavaScript per usare i file JSON, ma è utile conoscere il formato JSON perché è utilizzato da molte applicazioni web.

## Il modulo `csv`

Ogni riga in un file CSV rappresenta una riga nel foglio di calcolo e le virgole separano le celle nella riga.

Per esempio, il foglio *example.xlsx* scaricabile da <http://nostarch.com/automatedstuff/> avrebbe questo aspetto in un file CSV:

```
4/5/2015 13:34,Apples,73
4/5/2015 3:41,Cherries,85
4/6/2015 12:46,Pears,14
4/8/2015 8:59,Oranges,52
4/10/2015 2:07,Apples,152
```

4/10/2015 18:10,Bananas,23  
4/10/2015 2:40,Strawberries,98

Userò questo file per gli esempi di questo capitolo nella shell interattiva. Potete scaricare il file *example.csv* da <http://nostarch.com/automatestuff/> oppure inserire il testo in un editor e salvarlo come *example.csv*.

I **file CSV** sono semplici e sono privi di molte delle caratteristiche di un foglio di Excel. Per esempio,

- non hanno tipi per i valori, tutto è una stringa;
- non hanno impostazioni per dimensioni o colore del testo;
- non contengono più fogli di lavoro;
- non possono specificare larghezza e altezza delle celle;
- non possono avere celle unite;
- non possono avere incorporati immagini o grafici.

Il **vantaggio** dei file CSV è la loro **semplicità**. Sono ampiamente supportati da molti tipi di programmi, possono essere visualizzati negli editor di testo (compreso il file editor di IDLE) e costituiscono un modo immediato per rappresentare i dati dei fogli di calcolo. Il formato CSV è esattamente quel che dice il suo nome: un semplicemente **file di testo con valori separati da virgole**. Poiché i file CSV sono solo file di testo, potreste essere tentati di leggerli in una stringa e poi di elaborare quella stringa con le tecniche apprese nel [Capitolo 8](#). Per esempio, poiché ogni cella in un file CSV è separata da una virgola, potreste magari chiamare il metodo `split()` su ciascuna riga di testo per ottenere i valori. Non tutte le virgole in un file CSV però rappresentano il confine fra due celle: anche questi file hanno il loro gruppo di **caratteri escape** per consentire la presenza di virgole e altri caratteri all'interno dei valori. Il metodo `split()` non è in grado di gestire questi caratteri escape. Per evitare queste difficoltà, si usa sempre il modulo `csv` per leggere e scrivere file CSV.

## Oggetti Reader

Per **leggere dati** da un file CSV con il modulo `csv`, dovete creare un oggetto Reader. Un oggetto Reader permette di iterare sulle righe del file. Inserite quanto segue nella shell interattiva, con *example.csv* nella directory di lavoro corrente.

```
❶ >>> import csv
❷ >>> exampleFile = open('example.csv')
❸ >>> exampleReader = csv.reader(exampleFile)
❹ >>> exampleData = list(exampleReader)
❺ >>> exampleData
[['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'],
 ['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'],
 ['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'],
 ['4/10/2015 2:40', 'Strawberries', '98']]
```

Il modulo `csv` è incluso in Python, perciò possiamo importarlo in u senza doverlo prima installare. Per leggere un file CSV con il modulo `csv`, prima lo si apre con la funzione `open()` ❷, come si farebbe con qualsiasi altro file di testo, ma, anziché chiamare il metodo `read()` o `readlines()` sull'oggetto File

restituito da `open()`, lo si passa alla funzione `csv.reader()` ❸. Questa restituisce un oggetto Reader che potete usare. Notate che non si passa direttamente una stringa con il nome del file alla funzione `csv.reader()`.

Il procedimento più diretto per accedere ai valori nell'oggetto Reader consiste nel convertirlo in una semplice lista Python passandolo a `list()` ❹. L'uso di `list()` su questo oggetto Reader restituisce una lista di liste, che si può memorizzare in una variabile come `exampleData`. L'inserimento di `exampleData` nella shell visualizza la lista di liste ❺.

Ora che avete il file CSV come lista di liste, potete accedere al valore che si trova a una particolare riga e colonna con l'espressione `exampleData[row][col]`, dove `row` è l'indice di una delle liste in `exampleData` e `col` è l'indice dell'elemento che si vuole, all'interno di quella lista. Inserite quanto segue nella shell interattiva:

```
>>> exampleData[0][0]  
'4/5/2015 13:34'  
>>> exampleData[0][1]  
'Apples'  
>>> exampleData[0][2]  
'73'  
>>> exampleData[1][1]  
'Cherries'  
>>> exampleData[6][1]  
'Strawberries'
```

`exampleData[0][0]` entra nella prima lista e ci dà la prima stringa, `exampleData[0][2]` entra nella prima lista e ci dà la terza stringa, e via di questo passo.

## Leggere i dati da oggetti Reader in un ciclo for

Quando i file CSV sono di grandi dimensioni, si può usare l'oggetto Reader in un ciclo `for`, il che evita di dover caricare tutto il file in memoria in un colpo solo. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> import csv  
>>> exampleFile = open('example.csv')  
>>> exampleReader = csv.reader(exampleFile)  
>>> for row in exampleReader:  
    print('Row #' + str(exampleReader.line_num) + ' ' + str(row))  
Row #1 ['4/5/2015 13:34', 'Apples', '73']  
Row #2 ['4/5/2015 3:41', 'Cherries', '85']  
Row #3 ['4/6/2015 12:46', 'Pears', '14']  
Row #4 ['4/8/2015 8:59', 'Oranges', '52']  
Row #5 ['4/10/2015 2:07', 'Apples', '152']  
Row #6 ['4/10/2015 18:10', 'Bananas', '23']  
Row #7 ['4/10/2015 2:40', 'Strawberries', '98']
```

Dopo aver importato il modulo `csv` e creato un oggetto Reader dal file CSV, potete ciclare sulle righe nell'oggetto Reader. Ogni riga è una lista di valori, e ogni valore rappresenta una cella.

La chiamata alla funzione `print()` stampa il numero della riga corrente e i contenuti di quella riga. Per ottenere il numero della riga, si usa la variabile `line_num` dell'oggetto Reader, che contiene il numero della riga corrente.

Si può ciclare sull'oggetto Reader una sola volta. Per leggere nuvoamente il file CSV, bisogna chiamare csv.reader per creare un oggetto Reader.

## Oggetti Writer

Un oggetto Writer consente di **scrivere dati in un file CSV**. Per creare un oggetto Writer, si usa la funzione csv.writer(). Inserite quanto segue nella shell interattiva:

```
>>> import csv  
❶ >>> outputFile = open('output.csv', 'w', newline='')  
❷ >>> outputWriter = csv.writer(outputFile)  
>>> outputWriter.writerow(['spam', 'eggs', 'bacon', 'ham'])  
21  
>>> outputWriter.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])  
32  
>>> outputWriter.writerow([1, 2, 3.141592, 4])  
16  
>>> outputFile.close()
```

In primo luogo, si chiama `open()` e le si passa '`w`' per aprire un file in modalità scrittura ❶. Questo crea l'oggetto che si può poi passare a `csv.writer()` ❷ per creare un oggetto Writer.

In Windows, dovete passare anche una stringa vuota come argomento per parola chiave `newline` della funzione `open()`.

Per motivi tecnici, che vanno al di là delle finalità di questo libro, se vi dimenticate di impostare l'argomento `newline`, le righe in `output.csv` avranno righe bianche di troppo, come si vede nella [Figura 14.1](#).

	A	B	C	D	E	F	G
1	42	2	3	4	5	6	7
2							
3	2	4	6	8	10	12	14
4							
5	3	6	9	12	15	18	21
6							
7	4	8	12	16	20	24	28
8							
9	5	10	15	20	25	30	35
10							

**Figura 14.1** - Se vi dimenticate di passare l'argomento per parola chiave `newline=''` a `open()`, il file CSV avrà una riga vuota fra una riga di valori e l'altra.

Il metodo `writerow()` per gli oggetti Writer prende come argomento una lista. Ciascun valore nella lista viene collocato nella propria cella in file CSV di output. Il valore di ritorno di `writerow()` è il numero dei caratteri scritti nel file per quella riga (compresi i caratteri `newline`).

Questo codice produce un file `output.csv` che appare così:

```
spam,eggs,bacon,ham  
"Hello, world!",eggs,bacon,ham  
1,2,3.141592,4
```

Notate come l'oggetto `Writer` automaticamente effettui un escape della virgola nel valore 'Hello, world!', racchiudendola fra doppi apici nel file CSV. Il modulo `csv` vi risparmia di dover gestire appositamente questi casi speciali.

## Gli argomenti per parola chiave `delimiter` e `lineterminator`

Supponiamo che vogliate separare le celle con un carattere di tabulazione invece che con una virgola e che vogliate le righe con uno spazio fra una e l'altra. Potreste inserire qualcosa di simile a questo nella shell interattiva:

```
>>> import csv  
>>> csvFile = open('example.tsv', 'w', newline='')  
❶ >>> csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n')  
>>> csvWriter.writerow(['apples', 'oranges', 'grapes'])  
24  
>>> csvWriter.writerow(['eggs', 'bacon', 'ham'])  
17  
>>> csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])  
32  
>>> csvFile.close()
```

Questo cambia i caratteri che fungono da delimitatore e da terminatore di riga, rispettivamente. Il delimitatore è il carattere che compare fra una cella e l'altra di una riga. Per default, il delimitatore in un file CSV è una virgola. Il terminatore di riga è il carattere che si presenta alla fine di una riga e, per default, è un carattere newline. Potete modificare questi caratteri utilizzando gli argomenti per parola chiave `delimiter` e `lineterminator` con `csv.writer()`.

Passando `delimiter='\t'` e `lineterminator='\n\n'` **❶**, il carattere fra le celle diventa una tabulazione e il carattere a fine riga diventa due newline. Poi chiamiamo `writerow()` tre volte per avere altrettante righe.

Questo produce un file `example.tsv` con questi contenuti:

```
apples oranges grapes  
eggs bacon ham  
spam spam spam spam spam
```

Ora che le celle sono separate da tabulazioni, usiamo l'estensione del nome di file `.tsv`, che sta per “tab-separated values”.

## Progetto: eliminare l'intestazione da file CSV

Supponiamo che abbiate il noioso compito di eliminare la prima riga da varie centinaia di file CSV. Magari dovete poi darli in pasto a un processo automatizzato che vuole in ingresso solo i dati, senza le intestazioni delle colonne. Potreste aprire ciascun file in Excel, cancellare la prima riga e poi risalvare il file, ma ci vorrebbero ore. Scriviamo invece un programma che lo faccia per voi.

Il programma dovrà aprire tutti i file con estensione `.csv` nella directory di lavoro corrente, leggerne i contenuti e riscriverli senza la prima riga in un file con lo stesso nome. Questo sostituirà i vecchi contenuti del file CSV con i nuovi contenuti senza l'intestazione.

## NOTA

Come sempre, quando scrivete un programma che modifica file, usate la precauzione di effettuare prima un backup dei file, nella malaugurata eventualità che il programma non funzioni come vi aspettate. Non vorrete certo cancellare accidentalmente tutti i file originali.

Ad alto livello, il programma deve fare queste cose:

- trovare tutti i file CSV nella directory di lavoro corrente;
- leggere i contenuti di ciascun file;
- scrivere i contenuti, saltando la prima riga, in un nuovo file CSV.

A livello del codice, questo significa che il programma dovrà:

- ciclare su una lista di file fornita da `os.listdir()`, saltando i file non CSV;
- creare un oggetto Reader CSV e leggere i contenuti del file, utilizzando l'attributo `line_num` per stabilire quale riga saltare;
- creare un oggetto Writer CSV e scrivere i dati nel nuovo file.

Per questo progetto, apriete una nuova finestra di file editor e salvate il file con il nome `removeCsvHeader.py`.

### Passo 1: ciclare su ciascun file CSV

La prima cosa che il vostro programma deve fare è ciclare su una lista di tutti i nomi di file CSV presenti nella directory di lavoro corrente.

Cominciate in questo modo:

```
#! python3
# removeCsvHeader.py - Elimina l'intestazione da tutti I file CSV
# presenti nella directory di lavoro corrente.

import csv, os

os.makedirs('headerRemoved', exist_ok=True)

# Cicla su tutti i file nella directory di lavoro corrente.
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue # salta i file non-csv

    print('Sto eliminando l\'intestazione da ' + csvFilename + '...')

    # DA FARE: Leggere il file CSV (saltando la prima riga).

    # DA FARE: Scrivere il file CSV.
```

La chiamata a `os.makedirs()` creerà una cartella `headerRemoved` dove verranno scritti tutti i file CSV

senza intestazione. Un ciclo for su os.listdir('') farebbe fare un po' di strada, ma itererebbe su tutti i file nella directory di lavoro, perciò dovete aggiungere un po' di codice all'inizio del ciclo in modo da saltare i file il cui nome non finisce con .csv. L'enunciato continue ❶ fa in modo che il ciclo for passi al nome di file successivo, quando incontra un file che non è CSV.

Perché si veda qualcosa mentre il programma svolge il suo lavoro, stampate un messaggio che dice su quale file CSV il programma stia lavorando. Infine, alcuni commenti DA FARE indicano che cosa deve fare il resto del programma.

## Passo 2: leggere il file CSV

Il programma non elimina la prima riga dal file CSV, bensì crea una nuova copia del file CSV, priva della prima riga. Dato che il nome di file della copia è uguale a quello del file originale, la copia sovrascriverà l'originale.

Il programma dovrà avere un modo per tenere traccia del fatto che stia o meno iterando sulla prima riga. Aggiungete quanto segue a *removeCsvHeader.py*.

```
#! python3
# removeCsvHeader.py - Elimina l'intestazione da tutti I file CSV
# presenti nella directory di lavoro corrente.

--righe omesse--

# Legge il file CSV (saltando la prima riga).
csvRows = []
csvFileObj = open(csvFilename)
readerObj = csv.reader(csvFileObj)
for row in readerObj:
    if readerObj.line_num == 1:
        continue # salta la prima riga
    csvRows.append(row)
csvFileObj.close()

# DA FARE: Scrivere il file CSV.
```

Si può usare l'attributo `line_num` dell'oggetto Reader per stabilire quale riga del file CSV il programma stia leggendo. Un altro ciclo for itererà sulle righe restituite dall'oggetto CSV Reader e tutte le righe tranne la prima verranno accodate a `csvRows`.

Mentre il ciclo for itera su ciascuna riga, il codice verifica se `readerObj.line_num` è impostato a 1. Se è così, esegue un enunciato `continue` per passare alla riga successiva senza accodare questa a `csvRows`. Per ogni riga che segue la condizione sarà sempre `False` e la riga verrà accodata a `csvRows`.

## Passo 3: scrivere il file CSV senza la prima riga

Ora che `csvRows` contiene tutte le righe tranne la prima, la lista deve essere scritta in un file CSV nella cartella `headerRemoved`. Aggiungete quanto segue a *removeCsvHeader.py*:

```

#! python3
# removeCsvHeader.py - Elimina l'intestazione da tutti I file CSV
# presenti nella directory di lavoro corrente.

--righe omesse--
# Cicla su tutti i file nella directory di lavoro corrente.
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue # salta i file non-csv
--righe omesse--

# Scrive il file CSV.
csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w',
    newline='')
csvWriter = csv.writer(csvFileObj)
for row in csvRows:
    csvWriter.writerow(row)
csvFileObj.close()

```

L'oggetto Writer CSV scriverà la lista in un file CSV in *headerRemoved* con il nome csvFilename (che abbiamo usato anche con il Reader CSV). Questo sovrascriverà il file originale.

Una volta creato l'oggetto Writer, iteriamo sulla sottolista memorizzata in csvRows e scriviamo nel file ciascuna sottolista.

Dopo che questo codice è stato eseguito, il ciclo for esterno ❶ passerà al nome di file successivo in os.listdir('.'). Quando questo ciclo è finito, il programma è giunto al termine.

Per mettere alla prova il vostro programma, scaricate *removeCsvHeader.zip* da <http://nostarch.com/automatestuff/> ed estraete i contenuti in una cartella. Eseguite il programma *removeCsvHeader.py* in quella cartella. L'output sarà di questo tipo:

```

Sto eliminando l'intestazione da NAICS_data_1048.csv
Sto eliminando l'intestazione da NAICS_data_1218.csv
--righe omesse--
Sto eliminando l'intestazione da NAICS_data_9834.csv
Sto eliminando l'intestazione da NAICS_data_9986.csv

```

Il programma deve stampare un nome di file ogni volta che elimina la prima riga da un file CSV.

## Idee per programmi simili

I programmi che potreste scrivere per i file CSV sono simili a quelli che potreste scrivere per i file Excel, poiché si tratta pur sempre di file di fogli di calcolo. Potreste scrivere programmi, per esempio, che:

- confrontino i dati fra righe diverse in un file CSV o in più file diversi;
- copino dati specifici da un file CSV in un file Excel, o viceversa;
- verifichino se nei file CSV vi sono dati non valido o errori di formattazione e avvertano l'utente della eventuale presenza di errori;
- leggano dati da un file CSV come input per i vostri programmi Python.

## JSON e API

**JavaScript Object Notation** è un formato molto diffuso, che memorizza i dati come una singola stringa leggibile da un essere umano. JSON è il modo in cui i programmi JavaScript scrivono le loro strutture di dati e di solito è simile a quello che produrrebbe la funzione `pprint()` di Python. Non è necessario conoscere JavaScript per poter lavorare con dati inf ormato JSON.

Ecco un esempio di **dati formattati JSON**:

```
{"name": "Zophie", "isCat": true,  
"miceCaught": 0, "napsTaken": 37.5,  
"felineIQ": null}
```

Conoscere JSON è utile perché molti siti web offrono contenuti in questo formato per consentire l’interazione di programmi esterni con il sito. Si dice in questi casi che il sito fornisce una interfaccia per la **programmazione applicativa (API, application programming interface)**.

Accedere a una API è come accedere a qualsiasi altra pagina web attraverso un URL. La differenza è che i dati restituiti da una API sono formattati (con JSON, per esempio) per le macchine; le API non sono pensate per essere lette facilmente da esseri umani.

Molti **siti web** rendono disponibili i loro dati in formato JSON: Facebook, Twitter, Yahoo, Google, Tumblr, Wikipedia, Flickr, Data.gov, Reddit, IMDb, Rotten Tomatoes, LinkedIn sono tra i tanti che mettono a disposizione API che i programmi possono usare. Alcuni di questi siti richiedono una registrazione, che quasi sempre è gratuita. Dovrete trovare la documentazione per sapere quali URL il vostro programma deve interrogare per ottenere i dati che volete, nonché il **formato generale delle strutture di dati** JSON che restituiscono. Questa documentazione viene fornita normalmente dal sito che offre le sue API: se esiste una pagina per gli sviluppatori, quello è il posto in cui cercare la documentazione.

Utilizzando le API, potreste scrivere programmi che facciano cose come queste:

- estrarre dati grezzi dai siti web (accedere alle API spesso è molto più comodo che scaricare pagine web e analizzare l’HTML con Beautiful Soup);
- scaricare automaticamente i nuovi post da uno dei vostri account nei social network e pubblicarli su un altro account (per esempio, potreste prendere i post di Tumblr e pubblicarli su Facebook);
- creare una “encyclopedia dei film” per la vostra collezione personale di film, estraendo dati da IMDb, Rotten Tomatoes e Wikipedia e salvandoli in un singolo file di testo sul vostro computer.

Potete vedere alcuni esempi di API JSON nelle risorse elencate all’indirizzo <http://nostarch.com/automatestuff/>.

## Il modulo json

Il modulo `json` di Python gestisce tutti i dettagli della traduzione fra una stringa con dati JSON e valori Python per le funzioni `json.loads()` e `json.dumps()`. JSON non può memorizzare tutti i tipi di valori Python, ma solo valori dei tipi di dati seguenti: stringhe, interi, in virgola mobile, Booleani, liste, dizionari e `NoneType`. JSON non può rappresentare oggetti specifici di Python come oggetti `File`, oggetti `Reader` o `Writer CSV`, oggetti `Regex` o oggetti `WebElement` di Selenium.

## Leggere JSON con la funzione loads()

Per tradurre una stringa che contiene dati JSON in un valore Python, la si passa alla funzione `json.loads()`. (Il nome della funzione significa “carica stringa”.) Inserite quanto segue nella shell interattiva:

```
>>> stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0, "felineIQ": null}'  
>>> import json  
>>> jsonDataAsPythonValue = json.loads(stringOfJsonData)  
>>> jsonDataAsPythonValue  
{'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}
```

Dopo avere importato il modulo `json`, potete chiamare `loads()` e passargli una stringa di dati JSON. Notate che le stringhe JSON usano sempre i doppi apici. La funzione restituirà i dati sotto forma di dizionario Python. I dizionari non sono ordinati, perciò le coppie chiave-valore possono presentarsi in un ordine diverso, quando si stampa `jsonDataAsPythonValue`.

## Scrivere JSON con la funzione dumps()

La funzione `json.dumps()` traduce un valore Python in una stringa di dati formattati JSON. Inserite quanto segue nella shell interattiva:

```
>>> pythonValue = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}  
>>> import json  
>>> stringOfJsonData = json.dumps(pythonValue)  
>>> stringOfJsonData  
'{"isCat": true, "felineIQ": null, "miceCaught": 0, "name": "Zophie"}'
```

Il valore può essere solo uno dei seguenti tipi di dati fondamentali di Python: dizionario, lista, intero, virgola mobile, stringa, Booleano o `None`.

## Progetto: estrarre dati meteorologici

Controllare le condizioni del tempo sembra un’operazione abbastanza banale: aprite il browser web, fate clic sulla barra degli indirizzi, scrivete l’URL di un sito di previsioni meteorologiche (oppure ne cercate uno e poi fate clic sul link), aspettate che la pagina si carichi, scorrete oltre tutti gli annunci pubblicitari e così via.

In realtà, sono molti i passi noiosi che potreste saltare se aveste un programma in grado di scaricare le previsioni del tempo dei prossimi giorni e le stampasse come puro testo. Questo programma usa il modulo `requests` che abbiamo visto nel [Capitolo 11](#) per scaricare dati dal Web.

Nel complesso, il programma farà queste cose:

- legge la località richiesta dalla riga di comando;
- scarica dati meteorologici da [OpenWeatherMap.org](http://OpenWeatherMap.org);
- converte la stringa di dati JSON in una struttura di dati di Python;
- stampa le previsioni per oggi e per i prossimi due giorni.

Il codice quindi deve fare queste cose:

- concatenare le stringhe in sys.argv per ottenere la località;
- chiamare requests.get() per scaricare i dati sul tempo;
- chiamare json.loads() per convertire i dati JSON in una struttura di dati Python;
- stampare le previsioni del tempo.

Per questo progetto, aprite una nuova finestra di file editor e salvate il file con il nome *quickWeather.py*.

## Passo 1: prendere la località dall'argomento della riga di comando

L'input per il programma verrà dalla riga di comando. Cominciate a implementare *quickWeather.py* in questo modo:

```
#! python3
# quickWeather.py - Stampa le previsioni del tempo per una località indicata nella riga di comando.

import json, requests, sys

# Calcola la località dagli argomenti della riga di comando.
if len(sys.argv) < 2:
    print('Formato da usare: quickWeather.py località')
    sys.exit()
location = ' '.join(sys.argv[1:])

# DA FARE: Scaricare i dati JSON dalle API di OpenWeatherMap.org.

# DA FARE: Caricare i dati JSON in una variabile Python.
```

In Python, gli argomenti della riga di comando sono memorizzati nella lista sys.argv. Dopo la riga iniziale `#!` e gli enunciati di importazione, il programma verifica se gli argomenti della riga di comando siano più di uno. (Ricordate che sys.argv ha sempre almeno un elemento, sys.argv[0], che contiene il nome di file dello script Python.) Se nella lista vi è un solo elemento, allora l'utente non ha indicato sulla riga di comando una località e viene presentato un messaggio che precisa come debba essere usato il programma, prima che il programma stesso termini.

Gli argomenti della riga di comando sono separati da spazi. Un argomento come San Francisco, CA farebbe sì che sys.argv contenga `['quickWeather.py', 'San', 'Francisco', 'CA']`. Perciò dovete chiamare il metodo `join()` per unire tutte le stringhe tranne la prima in sys.argv. Memorizzate la stringa unita in una variabile location.

## Passo 2: scaricare i dati JSON

[OpenWeatherMap.org](http://OpenWeatherMap.org) presenta informazioni meteorologiche in tempo reale in formato JSON. Il vostro programma deve semplicemente scaricare la pagina all'indirizzo <http://api.openweathermap.org/data/2.5/forecast/daily?q=<Location>&cnt=3>, dove `<Location>` è il nome della città di cui volete le previsioni meteorologiche. Aggiungete quanto segue a *quickWeather.py*.

```
#! python3
# quickWeather.py – Stampa le previsioni del tempo per una località # indicata nella riga di comando.
```

--righe omesse--

```
# Scarica i dati JSON dalle API di OpenWeatherMap.org.
url ='http://api.openweathermap.org/data/2.5/forecast/daily?q=%s&cnt=3' % (location)
response = requests.get(url)
response.raise_for_status()
```

# DA FARE: Caricare i dati JSON in una variabile Python.

Abbiamo location dagli argomenti della riga di comando. Per creare l’URL a cui vogliamo accedere, usiamo il segnaposto %s e inseriamo la stringa memorizzata in location in quel punto della stringa URL. Memorizziamo il risultato in url e passiamo url a requests.get(). La chiamata requests.get() restituisce un oggetto Response, che potete esaminare per vedere se contiene errori chiamando raise\_for\_status(). Se non viene sollevata alcuna eccezione, il testo scaricato si troverà in response.text.

## Passo 3: caricare i dati JSON e stampare le previsioni

La variabile response.text contiene una lunga stringa di dati in formato JSON. Per convertirla in un valore Python, chiamate la funzione json.loads(). I dati JSON saranno qualcosa di simile a questi:

```
{'city': {'coord': {'lat': 37.7771, 'lon': -122.42},
           'country': 'United States of America',
           'id': '5391959',
           'name': 'San Francisco',
           'population': 0},
 'cnt': 3,
 'cod': '200',
 'list': [{['clouds': 0,
            'deg': 233,
            'dt': 1402344000,
            'humidity': 58,
            'pressure': 1012.23,
            'speed': 1.96,
            'temp': {'day': 302.29,
                      'eve': 296.46,
                      'max': 302.29,
                      'min': 289.77,
                      'morn': 294.59,
                      'night': 289.77},
            'weather': [{['description': 'sky is clear',
                         'icon': '01d'}]}]}
```

--righe omesse--

Potete vedere questi dati passando weatherData a pprint.pprint(). Potete visitare il sito <http://openweathermap.org/> per consultare altra documentazione sul significato di questi campi. Per esempio, la documentazione online vi dirà che il 302,29 dopo 'day' è la temperatura diurna in Kelvin, non in gradi centigradi o Fahrenheit.

Le informazioni che interessano vengono dopo 'main' e 'description'. Per stamparle in modo elegante, aggiungete quanto segue a *quickWeather.py*.

```

#! python3
# quickWeather.py - Stampa le previsioni del tempo per una località
# indicata nella riga di comando.

--righe omesse--

# Carica i dati JSON in una variabile Python.
weatherData = json.loads(response.text)

# Stampa le informazioni sul tempo.
❶ w = weatherData['list']
print('Current weather in %s:' % (location))
print(w[0]['weather'][0]['main'], '-', w[0]['weather'][0]['description'])
print()
print('Tomorrow:')
print(w[1]['weather'][0]['main'], '-', w[1]['weather'][0]['description'])
print()
print('Day after tomorrow:')
print(w[2]['weather'][0]['main'], '-', w[2]['weather'][0]['description'])

```

Notate come il codice memorizzi `weatherData['list']` nella variabile `w` per risparmiare un po' di battitura alla tastiera r. Si usano `w[0]`, `w[1]` e `w[2]` per recuperare i dizionari relativi al giorno corrente e ai due giorni successivi, rispettivamente. Ciascuno di questi dizionari ha una chiave 'weather', che contiene un valore lista. Siete interessati al primo elemento della lista, un dizionario annidato con parecchie altre chiavi, all'indice 0. Qui stampiamo i valori memorizzati nelle chiavi 'main' e 'description', separati tra loro da un trattino.

Quando questo programma viene eseguito da riga di comando con `quickWeather.py San Francisco, Ca`, l'output è qualcosa di analogo a questo:

Current weather in San Francisco, CA:  
Clear - sky is clear

Tomorrow:  
Clouds - few clouds

Day after tomorrow:  
Clear - sky is clear

## Idee per programmi simili

L'accesso ai dati meteo può costituire la base per molti tipi di programmi. Potete creare programmi simili che facciano le cose seguenti:

- raccogliere previsioni del tempo per vari campeggi o vari sentieri da percorrere a piedi, per vedere dove le condizioni meteorologiche siano migliori;
- pianificare un programma che verifichi regolarmente le previsioni meteorologiche e invii un avvertimento di gelo in arrivo, per poter spostare in casa i vasi con le piante (il [Capitolo 15](#) parla di pianificazione e il [Capitolo 16](#) spiega come spedire email);
- estrarre dati meteorologici da più siti per poterli vedere tutti insieme, o calcolare e presentare la media delle previsioni fornite da vari siti.

# Riepilogo

CSV e JSON sono formati comuni per la **memorizzazione dei dati in puro testo**. Sono facili da analizzare per i programmi pur essendo ancora leggibili da un essere umano, e per questo sono usati spesso per semplici fogli di calcolo o per i dati di applicazioni web. I moduli `csv` e `json` semplificano molto la lettura e la scrittura di file CSV e JSON.

Gli ultimi capitoli vi hanno mostrato come usare Python per analizzare informazioni estratte da file in molti formati diversi. Un compito comune è prendere dati da vari formati e analizzarli per estrarre le informazioni che servono. Questi compiti sono spesso così specifici che il software commerciale non è di aiuto nel modo migliore. Scrivendo i vostri programmi, potete fare in modo che il computer gestisca grandi quantità di dati presentati in questi formati.

Nel [Capitolo 15](#) ci allontaneremo dai formati dei dati e vedremo come fare in modo che i programmi comunichino con voi inviandovi email e sms.

## Domande di ripasso

1. Quali sono alcune caratteristiche dei fogli Excel che i fogli CSV non possiedono?
2. Che cosa si passa a `csv.reader()` e `csv.writer()` per creare oggetti Reader e Writer?
3. In quali modalità devono essere aperti gli oggetti File per gli oggetti Reader e Writer?
4. Quale metodo prende come argomento una lista e la scrive in un file CSV?
5. Che cosa fanno gli argomenti per parola chiave `delimiter` e `lineterminator`?
6. Quale funzione prende una stringa di dati JSON e restituisce una struttura di dati Python?
7. Quale funzione prende una struttura di dati Python e restituisce una stringa di dati JSON?

## Un po' di pratica

Per esercitarvi, scrivete un programma che svolga questa attività.

## Convertitore da Excel a CSV

Excel può salvare un foglio di calcolo in un file CSV con pochi clic del mouse, ma se vi capitasse di dover convertire centinaia di file Excel in CSV, vi ci vorrebbero ore di clic. Utilizzando il modulo `openpyxl` visto nel [Capitolo 12](#), scrivete un programma che legga tutti i file Excel presenti nella directory di lavoro corrente e li salvi come file CSV.

Un singolo file Excel può contenere più fogli di lavoro; dovrete creare un file CSV per ciascun foglio. I nomi dei file CSV dovranno essere `<nome di file excel>_<titolo del foglio>.csv`, dove `<nome di file excel>` è il nome del file Excel senza l'estensione (per esempio, 'spam\_data', non 'spam\_data.xlsx') e `<titolo del foglio>` è la stringa che si ottiene dalla variabile `title` dell'oggetto `Worksheet`. Questo programma comporterà vari cicli `for` annidati. Lo scheletro del programma sarà di questo tipo:

```
for excelFile in os.listdir('.'):
    # Salta i file non xlsx, carica l'oggetto workbook.
    for sheetName in wb.get_sheet_names():
        # Cicla su tutti i fogli nella cartella di lavoro.
        sheet = wb.get_sheet_by_name(sheetName)

        # Crea il nome del file CSV dal nome del file Excel e dal nome del foglio.
        # Crea l'oggetto csv.writer per questo file CSV.

    # Cicla su tutte le righe del foglio.
    for rowNum in range(1, sheet.get_highest_row() + 1):
        rowData = [] # append each cell to this list
        # Cicla su ogni cella della riga.
        for colNum in range(1, sheet.get_highest_column() + 1):
            # Accoda i dati di ciascuna cella a rowData.
        # Scrive la lista rowData nel file CSV.
    csvFile.close()
```

Scaricate il file ZIP *excelSpreadsheets.zip* dal sito <http://nostarch.com/automatestuff/>, estraete i fogli di calcolo nella stessa directory del programma. Potete usarli come file su cui mettere alla prova il vostro programma.

# Tenere traccia dell'ora, pianificare attività e lanciare programmi

Eseguire programmi mentre si è seduti al proprio computer va bene, ma è utile anche fare in modo che certi programmi vengano eseguiti senza la vostra supervisione diretta. **L'orologio** del computer può **pianificare i programmi** in modo che vengano eseguiti un certo giorno a una certa ora o a intervalli regolari. Per esempio, il vostro programma potrebbe accedere ogni ora a un sito web per verificare se vi sono variazioni, oppure svolgere un compito che richiede molta potenza di calcolo alle quattro del mattino mentre voi dormite. I **moduli time e datetime** di Python offrono le funzioni necessarie.

Potete anche scrivere programmi che lanciano altri programmi in giorni o ore prestabiliti utilizzando i moduli `subprocess` e `threading`. Spesso il modo più rapido per programmare è sfruttare le applicazioni che hanno già scritto altri.

## Il modulo time

L'**orologio di sistema** del vostro computer è regolato su una data, un'ora e un fuso orario specifici. Il modulo interno `time` consente ai programmi Python di leggere l'orologio per ricavarne l'ora. Le funzioni `time.time()` e `time.sleep()` sono le più utili del modulo `time`.

### La funzione `time.time()`

La **Unix epoch** (*epoca Unix*) è un riferimento temporale che si usa spesso nella programmazione: 12

AM dell'1 gennaio 1970, UTC (tempo coordinato universale, il fuso orario di riferimento). La funzione `time.time()` restituisce il numero di secondi trascorsi da quel momento, sotto forma di **valore in virgola mobile**. (Ricordate che un valore in virgola mobile è solo un numero con un punto decimale.) Questo numero è chiamato **epoch timestamp**. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> import time
```

```
>>> time.time()
```

1487180863.5389106

Qui ho chiamato `time.time()` il 15 febbraio 2017 alle 18.47.43 del fuso di Roma. Il valore restituito è il numero dei secondi trascorsi dall'epoca Unix e il momento in cui è stata chiamata `time.time()`.

## NOTA

Gli esempi della shell interattiva daranno date e ore del momento in cui questo capitolo è stato tradotto in italiano, nel febbraio 2017. Presumendo che non abbiate ancora imparato a viaggiare nel tempo, le date e gli orari che otterrete saranno diversi.

I timestamp possono essere usati per **profilare il codice**, cioè misura quanto tempo richiede l'esecuzione di un pezzo di codice.

Se chiamate `time.time()` all'inizio del blocco di codice che volete misurare e poi di nuovo alla fine, potete sottrarre il primo timestamp dal secondo e trovare il tempo trascorso fra le due chiamate. Per esempio, apriete una nuova finestra di file editor e inserite questo programma:

```
❶ import time
❷ def calcProd():
    #Calcola il prodotto dei primi 100.000 numeri.
    product = 1
    for i in range (1, 100000):
        product = product * i
    return product

❸ startTime = time.time()
❹ prod = calcProd()
❺ endTime = time.time()
❻ print('Il risultato è lungo %s cifre.' % (len(str(prod))))
❼ print('Per calcolarlo sono stati necessari %s secondi.' % (endTime - startTime))
```

In ❶, definiamo una funzione `calcProd()` che cicla sugli interi da 1 a 99999 e ne restituisce il prodotto. In ❷, chiamiamo `time.time()` e memorizziamo il valore in `startTime`. Subito dopo aver chiamato `calcProd()`, chiamiamo di nuovo `time.time()` e memorizziamo il valore in `endTime` ❸. Concludiamo stampando la lunghezza del prodotto restituito da `calcProd()` ❹ e il tempo che ha richiesto l'esecuzione di `calcProd()` ❽. Salvate questo programma come `calcProd.py` ed eseguitelo. L'output sarà simile a questo:

Il risultato è lungo 456569 cifre.

## NOTA

Un altro modo per profilare il codice è usare la funzione `cProfile.run()`, che offre un livello di dettaglio molto più ricco di informazioni rispetto alla semplice tecnica `time.time()`. La funzione `cProfile.run()` è spiegata all'indirizzo <http://docs.python.org/3/library/profile.html>.

## La funzione `time.sleep()`

Se avete bisogno di mettere in pausa per un po' il vostro programma, potete chiamare la funzione `time.sleep()` e passarle il numero dei secondi per cui volette che il programma rimanga in pausa. Inserite quanto segue nella shell interattiva:

```
>>> import time
>>> for i in range(3):
❶      print('Tic')
❷      time.sleep(1)
❸      print('Toc')
❹      time.sleep(1)

Tic
Toc
Tic
Toc
Tic
Toc
❺ >>> time.sleep(5)
```

Il ciclo `for` stamperà Tic ❶, aspetterà un secondo ❷, stamperà Toc ❸, aspetterà un secondo ❹, poi stamperà Tic, aspetterà, e così via fino a che Tic e Toc non sono stati stampati ciascuno tre volte.

La funzione `time.sleep()` bloccherà il programma (cioè non ritornerà e non consentirà al programma di eseguire altro codice) fino a che non è trascorso il numero di secondi passato a `time.sleep()`. Per esempio, se scrivete `time.sleep()` ❺, vedrete che il prompt (`>>>`) non comparirà di nuovo fino a che non sono trascorsi cinque secondi.

Tenete presente che la pressione di Ctrl-C non interrompe le chiamate `time.sleep()` in IDLE. IDLE attende per tutta la pausa prevista, prima di sollevare l'eccezione `KeyboardInterrupt`. Per aggirare il problema, anziché, poniamo, chiamare una volta `time.sleep(30)` per avere una pausa di 30 secondi, usate un ciclo `for` che effettui 30 chiamate a `time.sleep(1)`.

```
>>> for i in range(30):
    time.sleep(1)
```

Se premete Ctrl-C in qualche momento nel corso di quei 30 secondi, vedrete subito comparire l'eccezione `KeyboardInterrupt`.

## Arrotondare numeri

Quando si lavora con gli orari, si incontrano spesso **valori in virgola mobile** con molte cifre

decimali. Per semplificare il lavoro con questi valori, si può **abbreviarli** con la funzione interna di Python `round()`, che arrotonda un valore in virgola mobile al numero di cifre che si specifica. Si passano alla funzione il numero che si vuole arrotondare, più un secondo argomento facoltativo che rappresenta il numero delle cifre decimali a cui si vuole arrotondare. Se si omette il secondo argomento, `round()` arrotonda agli interi. Inserite quanto segue nella shell interattiva:

```
>>> import time  
>>> now = time.time()  
>>> now  
1425064108.017826  
>>> round(now, 2)  
1425064108.02  
>>> round(now, 4)  
1425064108.0178  
>>> round(now)  
1425064108
```

Dopo aver importato `time` e memorizzato `time.time()` in `now`, chiamiamo `round(now, 2)` per arrotondare `now` a due cifre decimali, `round(now, 4)` per arrotondare a quattro cifre decimali e `round(now)` per arrotondare agli interi.

## Progetto: supercronometro

Supponiamo che vogliate sapere quanto tempo perdete nello svolgere compiti noiosi che non avete ancora automatizzato. Non avete un cronometro fisico ed è stranamente difficile trovare una app gratuita per il portatile o lo smartphone che non sia piena di pubblicità e non invii una copia della cronologia del vostro browser agli inserzionisti. (Sta scritto nell'accordo di licenza a cui avete acconsentito. Avete letto l'accordo di licenza, vero?) Potete scrivere da soli un semplice programma cronometro in Python.

Ad alto livello, ecco che cosa farà il vostro programma:

- terrà traccia della quantità di tempo trascorsa fra una pressione del tasto Invio e l'altra, dove ogni pressione del tasto Invio fa partire una nuova “tappa” del timer;
- stampa il numero della tappa, il tempo totale e il tempo della tappa.

Questo significa che il vostro codice dovrà fare queste cose:

- stabilire l'ora corrente chiamando `time.time()` e memorizzarla come timestamp all'inizio del programma, così come all'inizio di ogni “tappa”;
- tenere un contatore delle tappe e incrementarlo ogni volta che l'utente preme Invio;
- calcolare il tempo trascorso sottraendo i timestamp;
- gestire l'eccezione `KeyboardInterrupt` in modo che l'utente possa premere Ctrl-C per uscire dal programma.

Aprite una nuova finestra di file editor e salvate il file con il nome `stopwatch.py`.

## Passo 1: impostare il programma perché tenga traccia del tempo

Il programma dovrà utilizzare l'ora corrente, perciò dovrete importare il modulo `time`. Il programma dovrà anche stampare delle brevi istruzioni per l'utente, prima di chiamare `input()`, perciò il timer può iniziare il suo lavoro dopo che l'utente preme Invio. Poi il codice comincerà a tener traccia del trascorrere del tempo.

Inserite il codice seguente del file editor, con un commento DA FARE come segnaposto per il resto del codice:

```
#! python3
# stopwatch.py – Un semplice cronometro.

import time

# Visualizza le istruzioni per l'uso del programma.
print('Premi Invio per iniziare. Poi, premi Invio ogni volta che vuoi fermare o far partire il cronometro. Premi Ctrl-C per finire.')
input() # Invio per iniziare
print('Iniziato.')
startTime = time.time() # prende l'ora di inizio della prima tappa
lastTime = startTime
lapNum = 1

# DA FARE: Iniziare a tracciare i tempi delle tappe.
```

Scritto il codice per visualizzare le istruzioni, bisogna iniziare la prima tappa, annotare il tempo e impostare a 1 il conteggio delle tappe.

## Passo 2: tracciare e stampare i tempi delle tappe

Ora scriviamo il codice per iniziare ciascuna nuova tappa, calcolare quanto tempo ha richiesto la precedente e calcolare il tempo totale trascorso dall'avvio del cronometro. Visualizzeremo il tempo della tappa e il tempo totale e incrementeremo il conteggio delle tappe per ogni nuova tappa. Aggiungete al vostro programma il codice seguente:

```
#! python3
# stopwatch.py – Un semplice cronometro.

import time

--righe omesse--

# Inizia a tracciare i tempi delle tappe.
❶ try:
❷     while True:
❸         input()
❹         lapTime = round(time.time() - lastTime, 2)
❺         totalTime = round(time.time() - startTime, 2)
❻         print('Lap #%s: %s (%s)' % (lapNum, totalTime, lapTime), end=' ')
❾         lapNum += 1
❿         lastTime = time.time() # reinizializza il tempo dell'ultima tappa
❬ except KeyboardInterrupt:
❭     # Gestisce l'eccezione Ctrl-C perché non visualizzi il suo messaggio di errore.
❮     print('\nFatto.')
```

Se l'utente preme Ctrl-C per fermare il cronometro, verrà sollevata l'eccezione `KeyboardInterrupt` e il programma andrà in crash in assenza di un enunciato `try`. Per impedire il crash, racchiudiamo questa parte del programma in un enunciato `try` ❶. Gestiremo l'eccezione nella clausola `except` ❷, in modo che, quando viene premuto Ctrl-C e l'eccezione viene sollevata, l'esecuzione del programma passa alla clausola `except` per stampare Fatto, invece del messaggio di errore `KeyboardInterrupt`. Finché questo non accade, l'esecuzione resta all'interno di un ciclo infinito ❸ che chiama `input()` e aspetta fino a che l'utente non preme Invio per concludere una tappa. Quando finisce una tappa, calcoliamo quanto tempo abbia richiesto sottraendo l'ora di inizio della tappa, `lastTime`, dall'ora corrente, `time.time()` ❹. Calcoliamo il tempo totale trascorso sottraendo l'ora di inizio generale, `startTime`, dall'ora corrente ❺. Poiché i risultati di questi calcoli avranno molte cifre decimali (per esempio 4.766272783279419), usiamo la funzione `round()` per arrotondare il valore in virgola mobile a due decimali in ❻ e ❼. In ❽, stampiamo il numero della tappa, il tempo totale trascorso e il tempo della tappa. Poiché la pressione del tasto Invio da parte dell'utente per la chiamata di `input()` stampa un a capo sullo schermo, passiamo `end="` alla funzione `print()` per evitare righe vuote. Dopo aver stampato le informazioni sulla tappa, ci prepariamo alla successiva sommando 1 al contatore `lapNum` e impostando `lastTime` all'ora corrente, che è l'ora di inizio della tappa successiva.

## Idee per programmi simili

Il tracciamento del tempo apre numerose possibilità per i vostri programmi. Anche se potete scaricare app per fare alcune di queste cose, il vantaggio di scrivervi da soli i vostri programmi è che saranno liberi e non infestati da pubblicità e funzioni prive di utilità. Potrete scrivere programmi analoghi per fare cose come queste:

- creare una semplice app “cartellino” che registri quando scrivete il nome di una persona e usi l'ora corrente per tener traccia degli orari di ingresso e uscita;
- aggiungere ai vostri programmi una funzionalità per visualizzare il tempo trascorso da quanto un processo è iniziato, per esempio un download che usa il modulo `requests` (vedete il [Capitolo 11](#));
- verificare periodicamente per quanto tempo è rimasto in esecuzione un programma e offrire all'utente la possibilità di interrompere attività che richiedono troppo tempo.

## Il modulo `datetime`

Il modulo `time` è utile per ottenere un timestamp (epoca Unix) con cui lavorare. Ma se volete visualizzare una data in un formato più comodo, o **fare calcoli con le date** (per esempio, stabilire che giorno era 205 giorni fa che giorno sarà fra 123 giorni), dovete usare il modulo `datetime`.

Il modulo `datetime` ha il proprio tipo di dati `datetime`. I valori `datetime` rappresentano un momento specifico nel tempo. Inserite quanto segue nella shell interattiva:

```
>>> import datetime
❶ >>> datetime.datetime.now()
❷ >>> datetime.datetime(2017, 2, 19, 11, 10, 49, 55, 53)
❸ >>> dt = datetime.datetime(2017, 2, 19, 16, 29, 0)
❹ >>> dt.year, dt.month, dt.day
(2017, 2, 19)
❺ >>> dt.hour, dt.minute, dt.second
(16, 29, 0)
```

La chiamata `datetime.datetime.now()` ❶ restituisce un oggetto `datetime` ❷ per la data e l'ora correnti, in base all'orologio del computer.

Questo oggetto include anno, mese, giorno, ora, minuto, secondo e microsecondo del momento corrente.

Potete anche recuperare un oggetto `datetime` per un momento specifico utilizzando la funzione `datetime.datetime()` ❸, passandole interi che rappresentano anno, mese, giorno, ora e secondo del momento che volete. Questi interi verranno memorizzati negli attributi `year`, `month`, `day` ❹, `hour`, `minute` e `second` ❻ dell'oggetto `datetime`.

Un timestamp (Unix epoch) può essere convertito in un oggetto `datetime` con la funzione `datetime.datetime.fromtimestamp()`. Data e ora dell'oggetto `datetime` verranno convertiti nel fuso orario locale. Inserite quanto segue nella shell interattiva:

```
>>> date time.datetime.fromtimestamp(1000000)
datetime.datetime(1970, 1, 12, 5, 46, 40)
>>> date time.datetime.fromtimestamp(time.time())
datetime.datetime(2017, 2, 16, 19, 33, 19, 75659)
```

Chiamando `datetime.datetime.fromtimestamp()` e passandole 1000000 si ottiene un oggetto `datetime` per il momento 1.000.000 di secondi dopo l'epoca Unix. Passando `time.time()`, il timestamp del momento corrente, si ottiene un oggetto `datetime` per il momento corrente. Perciò le espressioni `datetime.datetime.now()` e `datetime.datetime.fromtimestamp(time.time())` fanno la stessa cosa: danno un oggetto `datetime` per il momento presente.

Le indicazioni di tempo fornite da Python sono legate al fuso orario impostato sul computer: i vostri risultati potrebbero quindi essere diversi da quelli presentati negli esempi che seguono.

Gli oggetti `datetime` possono essere confrontati fra loro utilizzando gli operatori di confronto per stabilire quale precede l'altro. Un oggetto `datetime` più tardo ha il valore “più grande”. Inserite quanto segue nella shell interattiva:

```
❶ >>> halloween2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❷ >>> newyears2016 = datetime.datetime(2016, 1, 1, 0, 0, 0)
❸ >>> oct31_2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❹ >>> halloween2015 == oct31_2015
True
❺ >>> halloween2015 > newyears2016
False
❻ >>> newyears2016 > halloween2015
True
❼ >>> newyears2016 != oct31_2015
True
```

Qui si crea un oggetto `datetime` per il primo istante (mezzanotte) del 31 ottobre 2015 e lo si memorizza in `halloween2015` ❶.

Si crea un oggetto `datetime` per il primo istante dell'1 gennaio 2016 e lo si memorizza in `newyears2016` ❷. Poi si crea un altro oggetto per la mezzanotte del 31 ottobre 2015 e lo si memorizza in `oct31_2015`. Il confronto fra `halloween2015` e `oct31_2015` dice che sono uguali ❸.

Il confronto fra `newyears2016` e `halloween2015` mostra che il primo è maggiore di (cioè viene dopo) `halloween2015` ❹ ❽.

## Il tipo di dati timedelta

Il modulo `datetime` mette a disposizione anche un tipo di dati `timedelta`, che rappresenta un **periodo di tempo**, anziché un istante nel tempo. Inserite quanto segue nella shell interattiva:

```
❶ >>> delta = datetime.timedelta(days=11, hours=10, minutes=9, seconds=8)
❷ >>> delta.days, delta.seconds, delta.microseconds
(11, 36548, 0)
>>> delta.total_seconds()
986948.0
>>> str(delta)
'11 days, 10:09:08'
```

Per creare un oggetto `timedelta`, si usa la funzione `datetime.timedelta()`. La funzione `datetime.timedelta()` prende gli argomenti per parola chiave `weeks`, `days`, `hours`, `minutes`, `seconds`, `milliseconds` e `microseconds`. Non esistono argomenti per parola chiave `month` o `year` perché “un mese” o “un anno” sono periodi di tempo variabili a seconda del mese o dell’anno. Un oggetto `timedelta` ha la durata totale rappresentata in giorni, secondi e microsecondi. Questi numeri sono memorizzati negli attributi `days`, `seconds` e `microseconds`, rispettivamente. Il metodo `total_seconds()` restituisce la durata solo in numero di secondi. Passando un oggetto `timedelta` a `str()` si ottiene una rappresentazione a stringa dell’oggetto, elegantemente formattata.

In questo esempio, passiamo argomenti per parola chiave a `datetime.timedelta()` per specificare un periodo di 11 giorni, 10 ore, 9 minuti e 8 secondi, e memorizziamo l’oggetto `timedelta` restituito in `delta` ❶. L’attributo `days` dell’oggetto `timedelta` memorizza 11, e il suo attributo `seconds` memorizza 36548 (10 ore, 9 minuti e 8 secondi, espressi in secondi) ❷. La chiamata a `total_seconds()` ci dice che 11 giorni, 10 ore, 9 minuti e 8 secondi equivalgono a 986948 secondi. Infine, passando l’oggetto `timedelta` a `str()` si ottiene una stringa che spiega chiaramente la durata.

Si possono utilizzare gli operatori aritmetici per svolgere operazioni aritmetiche sulle date con i valori `datetime`. Per esempio, per calcolare la data a 1000 giorni da ora, potete inserire quanto segue nella shell interattiva:

```
>>> import datetime
>>> dt = datetime.datetime.now()
>>> dt
datetime.datetime(2017, 2, 17, 9, 24, 47, 284397)
>>> thousandDays = datetime.timedelta(days=1000)
>>> dt + thousandDays
datetime.datetime(2019, 11, 14, 9, 24, 47, 284397)
```

Come prima cosa, si crea un oggetto `datetime` per il momento corrente e lo si memorizza in `dt`. Poi si crea un oggetto `timedelta` per una durata di 1000 giorni e lo si memorizza in `thousandDays`. Si sommano poi `dt` e `thousandDays` per avere un oggetto `datetime` che corrisponde alla data mille giorni dopo il giorno corrente. Python fa i suoi calcoli e ci dice che 1000 giorni dopo il 27 febbraio 2017 sarà il 14 novembre 2019. Questo è utile quando si vuole calcolare una data di questo genere, perché altrimenti bisognerebbe tener conto di quanti giorni ci sono in un mese, degli anni bisestili e di altri particolari complicati: il modulo `datetime` invece gestisce tutto al posto nostro.

Gli oggetti `timedelta` possono essere sommati o sottratti a oggetti `datetime` o ad altri oggetti `timedelta` utilizzando gli operatori `+` e `-`. Un oggetto `timedelta` può essere moltiplicato o diviso per valori interi o in virgola mobile con gli operatori `*` e `/`. Inserite quanto segue nella shell interattiva:

```

❶ >>> oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)
❷ >>> aboutThirtyYears = datetime.timedelta(days=365 * 30)
>>> oct21st
datetime.datetime(2015, 10, 21, 16, 29)
>>> oct21st - aboutThirtyYears
datetime.datetime(1985, 10, 28, 16, 29)
>>> oct21st - (2 * aboutThirtyYears)
datetime.datetime(1955, 11, 5, 16, 29)

```

Qui abbiamo creato un oggetto `datetime` per il 21 ottobre 2015 ❶ e un oggetto `timedelta` per una durata di circa 30 anni (ipotizziamo che gli anni siano tutti di 365 giorni) ❷. Sottraendo `aboutThirtyYears` da `oct21st` otteniamo un oggetto `datetime` corrispondente alla data 30 anni prima del 21 ottobre 2015. Sottraendo `2 * aboutThirtyYears` da `oct21st` otteniamo un oggetto `datetime` per la data corrispondente a 60 anni prima del 21 ottobre 2015.

## Restare in attesa fino a una data specifica

Il metodo `time.sleep()` permette di **mettere in pausa** un programma per un certo numero di secondi. Utilizzando un ciclo `while`, potete mettere in pausa un programma fino a una data specifica. Per esempio, il codice seguente continuerà a ciclare fino a Halloween del 2018:

```

import datetime
import time
halloween2018 = datetime.datetime(2018, 10, 31, 0, 0, 0)
while datetime.datetime.now() < halloween2018:
    time.sleep(1)

```

La chiamata `time.sleep(1)` metterà in pausa il programma Python in modo che il computer non sprechi cicli di elaborazione solo per continuare a controllare l'ora. Il ciclo `while` si limiterà a verificare la condizione una volta al secondo e continuerà con il resto delle sue istruzioni dopo Halloween del 2018 (o quando lo programmate per fermarsi).

## Convertire in stringhe gli oggetti `datetime`

I timestamp e gli oggetti `datetime` non sono molto amichevoli per l'occhio umano. Utilizzate il metodo `strftime()` per visualizzare come stringa un oggetto `datetime`. (La *f* nel nome della funzione sta per “format”.)

Il metodo `strftime()` usa direttive simili alla formattazione delle stringhe di Python: sono tutte elencate nella [Tabella 15.1](#).

**Tabella 15.1** - Direttive `strftime()`.

Direttiva <code>strftime</code>	Significato
<code>%Y</code>	Anno con secolo, per esempio '2014'
<code>%y</code>	Anno senza secolo, da '00' a '99' (da 1970 a 2069)
<code>%m</code>	Mese come numero, da '01' a '12'
<code>%B</code>	Nome completo del mese, per esempio 'November'

%b	Nome abbreviato del mese, per esempio 'Nov'
%d	Giorno del mese, da '01' a '31'
%j	Giorno dell'anno, da '001' a '366'
%w	Giorno della settimana, da '0' (domenica) a '6' (sabato)
%A	Nome del giorno per esteso, per esempio 'Monday'
%a	Nome abbreviato del giorno, per esempio 'Mon'
%H	Ora (orologio di 24 ore), da '00' da '23'
%I	Hour (orologio di 12 ore), da '01' a '12'
%M	Minuto, da '00' a '59'
%S	Secondo, da '00' a '59'
%p	'AM' O 'PM'
%%	Carattere '%'

Passate a `strftime()` una stringa di formato personalizzata contenente le direttive di formattazione (e i trattini, le virgole ecc. che desiderate) e `strftime()` vi restituirà le informazioni dell'oggetto `datetime` come una stringa formattata. Inserite quanto segue nella shell interattiva:

```
>>> oct21st = datetime.datetime(2017, 10, 21, 16, 29, 0)
>>> oct21st.strftime('%Y/%m/%d %H:%M:%S')
'2017/10/21 16:29:00'
>>> oct21st.strftime('%I:%M %p')
'04:29 PM'
>>> oct21st.strftime("%B of '%y")
"October of '17"
```

Qui abbiamo un oggetto `datetime` per il 21 ottobre 2017 alle 16:29 del pomeriggio, memorizzato in `oct21st`. Passando alla funzione `strftime()` la stringa di formato personalizzato `'%Y/%m/%d %H.%M.%S'` otteniamo di ritorno una stringa che contiene 2017, 10 e 21 separati da barre e 17, 29 e 00 separati da virgole. Passando `'%I.%M %p'` si ottiene '04:29 PM' (3) e passando `"%B of '%y"` si ottiene "October of '17". Notate che `strftime()` non inizia con `datetime.datetime`.

## Convertire stringhe in oggetti datetime

Se avete una stringa di informazioni su una data, come '2015/10/21 16:29:00' o 'October 21, 2015', e dovete convertirla in un oggetto `datetime`, usate la funzione `datetime.datetime.strptime()`, che è l'inversa del metodo `strftime()`. Bisogna passare una stringa di formato personalizzato utilizzando le stesse direttive di `strftime()`, in modo che `strptime()` sappia come analizzarla e comprenderla. (La *p* nel nome della funzione sta per *parse*.)

Inserite quanto segue nella shell interattiva:

```
❶ >>> datetime.datetime.strptime('October 21, 2017', '%B %d, %Y')
datetime.datetime(2017, 10, 21, 0, 0)
>>> datetime.datetime.strptime('2017/10/21 16:29:00', '%Y/%m/%d %H:%M:%S')
datetime.datetime(2017, 10, 21, 16, 29)
>>> datetime.datetime.strptime("October of '17", "%B of '%y")
datetime.datetime(2017, 10, 1, 0, 0)
>>> datetime.datetime.strptime("November of '63", "%B of '%y")
datetime.datetime(2063, 11, 1, 0, 0)
```

Per ottenere un oggetto `datetime` dalla stringa 'October 21, 2017', si passa la stringa come primo argomento a `strptime()` e la stringa di formato personalizzato corrispondente come secondo argomento ❷. La stringa con le informazioni sulla data deve corrispondere esattamente alla stringa di formato personalizzato, altrimenti Python solleverà un'eccezione `ValueError`.

## Riepilogo delle funzioni di data e ora di Python

Date e ore in Python possono coinvolgere parecchi tipi di dati diversi e molte funzioni diverse. Ecco una sintesi dei tre diversi tipi di valori utilizzati per rappresentare l'ora.

- Un timestamp “epoca Unix” (utilizzato dal modulo `time`) è un valore in virgola mobile o intero che dà il numero dei secondi trascorsi dalle 12 AM dell’1 gennaio 1970, UTC.
- Un oggetto `datetime` (del modulo `datetime`) comprende interi memorizzati negli attributi `year`, `month`, `day`, `hour`, `minute` e `second`.
- Un oggetto `timedelta` (del modulo `datetime`) rappresenta un periodo di tempo, anziché un istante specifico.

Queste sono invece le funzioni di data e ora, con relativi parametri e valori di ritorno.

- La funzione `time.time()` restituisce un valore in virgola mobile con il timestamp del momento corrente.
- La funzione `time.sleep(secondi)` mette in pausa il programma per la quantità di `secondi` specificata dall’argomento `secondi`.
- La funzione `datetime.datetime(year, month, day, hour, minute, second)` restituisce un oggetto `datetime` del momento specificato dagli argomenti. Se non vengono forniti gli argomenti `hour`, `minute` o `second`, assumono per default valore 0.
- La funzione `datetime.datetime.now()` restituisce un oggetto `datetime` del momento corrente.
- La funzione `datetime.datetime.fromtimestamp(epoca)` restituisce un oggetto `datetime` del momento rappresentato dall’argomento `timestamp` `epoca`.
- La funzione `datetime.timedelta(days, hours, minutes, seconds, milliseconds, microseconds)` restituisce un oggetto `timedelta` che rappresenta un periodo di tempo. Gli argomenti per parola chiave della funzione sono tutti opzionali e non comprendono `month` o `year`).
- Il metodo `total_seconds()` per gli oggetti `timedelta` restituisce il numero di secondi che l’oggetto `timedelta` rappresenta.
- Il metodo `strftime(format)` restituisce una stringa con la data e l’ora rappresentate dall’oggetto `datetime` in un formato personalizzato basato sulla stringa `format`. Vedete la [Tabella 15.1](#) per i particolari dei formati.
- La funzione `datetime.datetime.strptime(time_string, format)` restituisce un oggetto `datetime` del momento

specificato da `time_string`, analizzata in base all'argomento stringa `format`. Vedete la [Tabella 15.1](#) per i particolari dei formati.

## Multithreading

Per introdurre il concetto di **multithreading**, facciamo un esempio. Supponiamo che vogliate pianificare l'esecuzione di un certo codice dopo un dato intervallo o a un'ora specifica. Potreste aggiungere all'inizio del programma qualcosa del genere:

```
import time, datetime

startTime = datetime.datetime(2029, 10, 31, 0, 0, 0)
while datetime.datetime.now() < startTime:
    time.sleep(1)

print('Il programma ora parte a Halloween 2029')
--altre righe di codice--
```

Questo codice precisa una data di avvio, il 31 ottobre 2029, e continua a chiamare `time.sleep(1)` finché non arriva quel momento. Il programma non può fare nulla mentre aspetta che le chiamate del ciclo `time.sleep()` finiscano: se ne sta lì in attesa fino a Halloween del 2029. Questo perché i programmi Python per impostazione definita hanno un solo thread di esecuzione.

Per capire che cos'è un **thread di esecuzione**, ripensate a quando abbiamo parlato nel [Capitolo 2](#) di **controllo del flusso**: abbiamo paragonato l'esecuzione di un programma al mettere un dito su una riga di codice nel programma e allo spostare il dito sulla riga successiva o dove viene mandato da un enunciato di controllo del flusso. Un programma a thread singolo ha un solo dito, ma un programma multithread ha più dita, e ciascun dito continua a passare alla riga di codice successiva definita dagli enunciati di controllo del flusso, ma le dita possono essere in punti diversi del programma ed eseguire contemporaneamente righe di codice diverse. (Fin qui tutti i programmi che abbiamo visto erano a thread singolo.)

Anziché lasciare che tutto il codice attenda la fine della funzione `time.sleep()`, potete eseguire il codice ritardato o pianificato in un thread distinto, utilizzando il modulo `threading` di Python. Il thread separato rimarrà in pausa in attesa della fine delle chiamate a `time.sleep()`, ma nel frattempo il vostro programma potrà svolgere qualche altra attività nel thread originale.

Per creare un thread distinto, dovete prima creare un oggetto `Thread` chiamando la funzione `threading.Thread()`. Inserite il codice seguente in un nuovo file e salvate con il nome `threadDemo.py`:

```
❶ import threading, time
    print('Inizio del programma.')
❷ def takeANap():
        time.sleep(5)
        print('Sveglia!')
❸ threadObj = threading.Thread(target=takeANap)
❹ threadObj.start()

    print('Fine del programma.')
```

In ❶, definiamo una funzione che vogliamo usare in un nuovo thread. Per creare un oggetto `Thread`,

chiamiamo `threading.Thread()` e le passiamo l'argomento per parola chiave `target=takeANap()` ❷. Questo significa che la funzione che vogliamo chiamare nel nuovo thread è `takeANap()`. Notate che l'argomento per parola chiave è `target=takeANap`, non `target=takeANap()`; questo perché si vuole passare come argomento la funzione `takeANap()` stessa, non si vuole chiamare `takeANap()` e passare il valore che restituisce.

Dopo aver memorizzato in `threadObj` l'oggetto `Thread` creato da `threading.Thread()`, chiamiamo `threadObj.start()` ❸ per creare il nuovo thread e iniziare a eseguire la funzione `target` nel nuovo thread.

Quando il programma viene eseguito, l'output sarà come questo:

```
Inizio del programma.
```

```
Fine del programma.
```

```
Sveglia!
```

Siete perplessi? Se `print('Fine del programma.')` è l'ultima riga del programma, potreste pensare che sia l'ultima cosa che verrà stampata. Il motivo per cui `Sveglia!` viene dopo è che, quando viene chiamato `threadObj.start()`, la funzione `target` per `threadObj` viene eseguita in un nuovo thread. Potete pensarla come un secondo dito che compare all'inizio della funzione `takeANap()`. Il thread principale continua con `print('Fine del programma.')`. Nel frattempo, il nuovo thread, che ha iniziato a eseguire la chiamata `time.sleep(5)` rimane in pausa per 5 secondi; quando si sveglia dal suo pisolino di cinque secondi, stampa '`Sveglia!`' e quindi ritorna dalla funzione `takeANap()`. Cronologicamente, '`Sveglia!`' è l'ultima cosa stampata dal programma.

Normalmente un programma termina quando viene eseguita l'ultima riga del codice nel file (o è stata chiamata la funzione `sys.exit()`), ma `threadDemo.py` ha due thread. Il primo è il thread originale, avviato all'inizio del programma, che termina dopo `print('Fine del programma.')`. Il secondo thread viene creato quando si chiama `threadObj.start()`, inizia all'inizio della funzione `takeANap()` e finisce dopo che `takeANap()` ritorna.

Un programma Python non termina finché non sono terminati tutti i suoi thread. Quando eseguite `threadDemo.py`, anche se il thread originale è terminato, il secondo thread sta eseguendo ancora la chiamata `time.sleep(5)`.

## Passaggio di argomenti alla funzione target del thread

Se la funzione `target` che si vuole eseguire nel nuovo thread prende argomenti, li si può passare a `threading.Thread()`. Per esempio, supponiamo di voler eseguire questa chiamata `print()` in un suo thread:

```
>>> print('Cats', 'Dogs', 'Frogs', sep=' & ')
Cats & Dogs & Frogs
```

Questa chiamata `print()` ha tre argomenti regolari, '`Cats`', '`Dogs`' e '`Frogs`' e un argomento per parola chiave, `sep=' & '`. Gli argomenti regolari possono essere passati sotto forma di lista all'argomento per parola chiave `args` in `threading.Thread()`. L'argomento per parola chiave può essere specificato come dizionario all'argomento per parola chiave `kwargs` in `threading.Thread()`. Inserite quanto segue nella shell interattiva:

```
>>> import threading
>>> threadObj = threading.Thread(target=print, args=['Cats', 'Dogs', 'Frogs'],
>>>                               kwargs={'sep': ' & '})
>>> threadObj.start()
Cats & Dogs & Frogs
```

Per essere sicuri che gli argomenti 'Cats', 'Dogs' e 'Frogs' vengano passati a `print()` nel nuovo thread, passiamo `args=['Cats', 'Dogs', 'Frogs']` a `threading.Thread()`. Per essere sicuri che l'argomento per parola chiave `'sep': ' & '` venga passato a `print()` nel nuovo thread, passiamo `kwargs={'sep': ' & '}` a `threading.Thread()`. La chiamata `threadObj.start()` crea un nuovo thread per chiamare la funzione `print()` e le passa 'Cats', 'Dogs' e 'Frogs' come argomenti e `'sep': ' & '` come argomento per parola chiave `sep`. Questo è invece un modo errato di creare il nuovo thread che chiama `print()`:

```
threadObj = threading.Thread(target=print('Cats', 'Dogs', 'Frogs', sep=' & '))
```

Quel che succede in questo caso è che viene chiamata la funzione `print()` e che si passa il suo valore di ritorno (che è sempre `None`) come argomento per parola chiave `target`. Non viene passata la funzione `print()` stessa. Quando si passano degli argomenti a una funzione in un nuovo thread, bisogna usare gli argomenti per parola chiave `args` e `kwargs` della funzione `threading.Thread()`.

## Problemi di concorrenza

Si possono creare facilmente molti nuovi thread e li si può eseguire tutti contemporaneamente. I thread multipli però possono anche creare problemi di **concorrenza**, problemi che si presentano quando i thread leggono e scrivono variabili contemporaneamente, intralciandosi a vicenda. I problemi di concorrenza possono essere difficili da riprodurre in modo regolare, il che ne rende difficile la soluzione.

La programmazione multithread è un tema molto ampio, che va al di là delle finalità di questo libro. Dovete ricordare questo: per evitare i problemi di concorrenza, fate in modo che i vari thread non leggano o scrivano le stesse variabili. Quando create un nuovo oggetto `Thread`, assicuratevi che la sua funzione `target` usi solo variabili locali a quella funzione. In questo modo eviterete di incorrere in problemi di concorrenza difficili da risolvere.

### NOTA

Un tutorial sulla programmazione multithread per chi è alle prime armi si trova all'indirizzo <http://nostarch.com/automatestuff/>.

## Progetto: scaricamento multithread da XKCD

Nel [Capitolo 11](#), avete scritto un programma che scaricava tutte le strisce a fumetti dal sito XKCD. Si trattava di un programma a thread singolo: scaricava una striscia alla volta. Gran parte del tempo di esecuzione del programma veniva spesa nello stabilire la connessione per iniziare il download e poi per scrivere su disco fisso le immagini scaricate. Se avete una connessione Internet a banda larga, il programma a thread singolo non sfruttava a pieno la banda disponibile.

Un programma multithread in cui alcuni thread scaricano le strisce mentre altri stabiliscono le connessioni e scrivono i file di immagine su disco sfrutta la connessione Internet con maggiore efficienza e scarica la collezione di strisce più rapidamente. Apriate una nuova finestra di file editor e salvate il file con il nome `multidownloadXkcd.py`. Modificherete il programma in modo da aggiungere il multithreading. Il codice sorgente modificato può essere scaricato da

## Passo 1: modificare il programma in modo da usare una funzione

Questo programma sarà per gran parte uguale al codice del Capitolo 11, perciò non ripeterò le spiegazioni per il codice relativo a Requests e BeautifulSoup. I cambiamenti principali che dovrete apportare riguardano l'importazione del modulo `threading` e la creazione della funzione `downloadXkcd()`, che prende come parametri i numeri iniziale e finale delle strisce da scaricare.

Per esempio, chiamando `downloadXkcd(140, 280)` si farà partire un ciclo per scaricare le strisce agli indirizzi a <http://xkcd.com/140>, <http://xkcd.com/141>, <http://xkcd.com/142> e così via, fino a <http://xkcd.com/279>.

Ciascun thread che creerete chiamerà `downloadXkcd()` e le passerà un intervallo diverso di strisce da scaricare.

Aggiungete il codice seguente al vostro programma *multidownloadXkcd.py*.

```
#! python3
# multidownloadXkcd.py - Scarica i fumetti XKCD usando più thread.

❶ import requests, os, bs4, threading
❷ os.makedirs('xkcd', exist_ok=True) # salva le strisce in ./xkcd

❸ def downloadXkcd(startComic, endComic):
❹     for urlNumber in range(startComic, endComic):
❺         # Scarica la pagina.
❻         print('Downloading page http://xkcd.com/%s...' % (urlNumber))
❼         res = requests.get('http://xkcd.com/%s' % (urlNumber))
⪻         res.raise_for_status()

⪼         soup = bs4.BeautifulSoup(res.text)

⪽         # Trova l'URL dell'immagine del fumetto.
⪾         comicElem = soup.select('#comic img')
⪿         if comicElem == []:
⪰             print('Non sono riuscito a trovare l\'immagine della striscia.')
⪱         else:
⪲             comicUrl = comicElem[0].get('src')

⪳             # Scarica l'immagine.
⪴             print('Sto scaricando l\'immagine %s...' % (comicUrl))
⪵             res = requests.get(comicUrl)
⪶             res.raise_for_status()

⪷             # Salva l'immagine in ./xkcd
⪸             imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)), 'wb')
⪹             for chunk in res.iter_content(100000):
⪺                 imageFile.write(chunk)
⪻             imageFile.close()

⪼ # DA FARE: Creare e avviare gli oggetti Thread.

⪽ # DA FARE: Attendere la conclusione di tutti i thread.
```

Dopo aver importato i moduli che ci servono, creiamo una directory in cui memorizzare le strisce ❶ e iniziamo a definire `downloadXkcd` ❷. Cicliamo su tutti i numeri nell'intervallo specificato ❸ e scarichiamo ciascuna pagina ❹. Usiamo BeautifulSoup per esplorare l'HTML di ciascuna pagina ❺ e trovare l'immagine della striscia ❻. Se su una pagina non viene trovata alcuna immagine, stampiamo un messaggio, altrimenti otteniamo l'URL dell'immagine ⪻ e la scarichiamo ⪼. Infine, salviamo l'immagine nella directory che abbiamo creato.

## Passo 2: creare e avviare i thread

Ora che abbiamo definito `downloadXkcd()`, creiamo i vari thread, ciascuno dei quali chiamerà `downloadXkcd()` per scaricare un diverso gruppo di strisce dal sito XKCD.  
Aggiungete il codice seguente a `multidownloadXkcd.py`, dopo la definizione della funzione `downloadXkcd()`:

```
#! python3
# multidownloadXkcd.py - Scarica i fumetti XKCD usando più thread.

--righe omesse--

# Crea e avvia gli oggetti Thread.
downloadThreads = [] # a list of all the Thread objects
for i in range(0, 1400, 100): # loops 14 times, creates 14 threads
    downloadThread = threading.Thread(target=downloadXkcd, args=(i, i + 99))
    downloadThreads.append(downloadThread)
    downloadThread.start()
```

Per prima cosa creiamo una lista vuota `downloadThreads`; la lista ci aiuterà a tener traccia dei molti oggetti `Thread` che creeremo. Poi iniziamo il nostro ciclo `for`. Ogni volta che passiamo nel ciclo, creiamo un oggetto `Thread` con `threading.Thread()`, accodiamo l'oggetto `Thread` alla lista e chiamiamo `start()` per avviare l'esecuzione di `downloadXkcd` nel nuovo thread. Dato che il ciclo `for` imposta la variabile `i` da 0 a 1400 in passi di 100, `i` verrà impostata a 0 alla prima iterazione, a 100 nella seconda iterazione, a 200 nella terza e così via. Poiché passiamo a `threading.Thread args=(i, i + 99)`, i due argomenti passati a `downloadXkcd` saranno 0 e 99 nella prima iterazione, 100 e 199 nella seconda, 200 e 299 nella terza e così via.

Quando viene chiamato il metodo `start()` dell'oggetto `Thread` e il nuovo thread inizia a eseguire il codice all'interno di `downloadXkcd()`, il thread principale continuerà con l'iterazione successiva del ciclo `for` e creerà il thread successivo.

## Passo 3: attendere la conclusione di tutti i thread

Il thread principale continua normalmente, mentre gli altri thread creati scaricano le strisce. Ma poniamo che vi sia del codice che non volete eseguire nel thread principale fino a che non sono completati tutti gli altri thread. La chiamata al metodo `join()` di un oggetto `Thread` imporrà un blocco fino a che quel thread non è completato. Con un ciclo `for` che itera su tutti gli oggetti `Thread` nella lista `downloadThreads`, il thread principale può chiamare il metodo `join()` su ciascuno degli altri thread. Aggiungete quanto segue in fondo al vostro programma:

```
#! python3
# multidownloadXkcd.py - Scarica i fumetti XKCD usando più thread.
```

--righe omesse--

```
# Attende la conclusione di tutti i thread.
for downloadThread in downloadThreads:
    downloadThread.join()
print('Fatto.')
```

La stringa 'Fatto.' non verrà stampata fino a che non sono ritornate tutte le chiamate `join()`. Se un oggetto Thread ha già completato il suo lavoro quando viene chiamato il suo metodo `join()`, il metodo non farà altro che ritornare immediatamente. Se volete estendere questo programma con codice che venga eseguito solo dopo che sono state scaricate tutte le strisce, potete sostituire la riga `print('Fatto.')` con il vostro nuovo codice.

## Lanciare altri programmi da Python

Un programma Python può lanciare altri programmi sul vostro computer con la funzione `Popen()` nel modulo interno `subprocess`. (La *P* nel nome della funzione sta per *Process*.) Se avete aperto più istanze di un'applicazione, ciascuna di esse è un processo distinto dello stesso programma. Per esempio, se aprite più finestre del vostro browser contemporaneamente, ciascuna è un processo distinto del programma browser. La [Figura 15.1](#) mostra, a titolo di esempio, più processi calcolatrice aperti contemporaneamente.



**Figura 15.1** - Sei processi in esecuzione contemporanea dello stesso programma calcolatrice.

Ogni processo può avere più thread. A differenza dei thread, un processo non può leggere e scrivere direttamente le variabili di un altro processo. Se pensate un programma multithread come un programma con molte dita che seguono il codice sorgente, avere aperti più processi dello stesso programma è come avere tanti amici diversi con una loro copia del codice sorgente del programma. Tutti eseguono indipendentemente lo stesso programma.

Se volete avviare un programma esterno dal vostro script Python, passate il nome di file del programma a `subprocess.Popen()`. (In Windows, fate clic destro sulla voce relativa all'applicazione nel

menu **Start** e selezionate **Proprietà** per vedere il nome di file dell'applicazione. In OS X, fate Ctrl-clic sull'applicazione e selezionate **Mostra contenuti pacchetto** per trovare il percorso del file eseguibile.) La funzione `Popen()` ritornerà immediatamente. Ricordate che il programma lanciato non viene eseguito nello stesso thread del programma Python.

Su un computer Windows, inserite per esempio quanto segue nella shell interattiva:

```
>>> import subprocess  
>>> subprocess.Popen('C:\\Windows\\System32\\calc.exe')  
<subprocess.Popen object at 0x000000003055A58>
```

In Ubuntu Linux, inserite invece:

```
>>> import subprocess  
>>> subprocess.Popen('/usr/bin/gnome-calculator')  
<subprocess.Popen object at 0x7f2bcf93b20>
```

Sotto OS X, il procedimento è leggermente diverso. Consultate la sezione “[Aprire file con applicazioni predefinite](#)” a pagina 365.

Il valore di ritorno è un oggetto `Popen`, che ha due metodi utili: `poll()` e `wait()`.

Potete pensare che il metodo `poll()` chieda all'amico se ha finito di eseguire il codice che gli è stato dato. Il metodo `poll()` restituirà `None` se il processo è ancora in esecuzione quando viene chiamata `poll()`. Se il programma è terminato, restituirà un intero, il **codice di uscita** (*exit code*) del processo. Il codice di uscita è utilizzato per indicare se il processo è terminato senza errori (codice 0) o se un errore ha portato alla chiusura del processo (un codice diverso da 0; in genere è 1, ma dipende dal programma).

Il metodo `wait()` ha la stessa funzione dell'aspettare che l'amico abbia finito di lavorare sul suo codice prima di continuare a lavorare sul vostro. Il metodo `wait()` bloccherà l'esecuzione finché il processo lanciato non è terminato. Questo è utile se volete che il vostro programma rimanga in pausa fino a quando l'utente non conclude il lavoro con l'altro programma. Il valore restituito da `wait()` è l'intero che rappresenta il codice di uscita del processo.

In Windows, inserite quanto segue nella shell interattiva. Notate che la chiamata `wait()` bloccherà il programma fino a che non chiudete il programma calcolatrice che avete lanciato.

```
❶ >>> calcProc = subprocess.Popen('c:\\Windows\\System32\\calc.exe')  
❷ >>> calcProc.poll() == None  
True  
❸ >>> calcProc.wait()  
0  
>>> calcProc.poll()  
0
```

Qui si apre un processo calcolatrice ❶. Mentre è ancora in esecuzione, verifichiamo se `poll()` restituisce `None` ❷. Dovrebbe essere così, perché il processo è ancora in esecuzione. Poi chiudiamo il programma calcolatrice e chiamiamo `wait()` sul processo terminato ❸. `wait()` e `poll()` ora restituiscono 0, il che indica che il processo è terminato senza errori.

## Passaggio di argomenti da riga di comando a `Popen()`

Si possono passare argomenti da riga di comando ai processi che si creano con `Popen()`. Per farlo, si passa a `Popen()` come unico argomento una lista. La prima stringa nella lista sarà il nome del file eseguibile del programma che si vuole lanciare; le stringhe successive saranno gli argomenti da riga di comando da passare al programma quando si avvia. In effetti, questa lista sarà il valore di `sys.argv` per il programma lanciato.

La maggior parte delle applicazioni con un'**interfaccia utente grafica (GUI)** non usano argomenti da riga di comando come i programmi basati su terminale o su riga di comando; la maggior parte delle applicazioni GUI accetterà però un unico argomento per un file che l'applicazione aprirà immediatamente quando viene lanciata. Per esempio, se usate Windows, create un semplice file di testo con il nome `C:\hello.txt` e poi inserite quanto segue nella shell interattiva:

```
>>> subprocess.Popen(['C:\\Windows\\notepad.exe', 'C:\\hello.txt'])  
<subprocess.Popen object at 0x0000000032DCEB8>
```

Questo non solo lancerà l'applicazione Blocco note, ma aprirà immediatamente anche il file `C:\hello.txt`.

## Pianificazione attività, launchd e cron

Probabilmente conoscerete Pianificazione attività in Windows, launchd in OS X o il pianificatore cron di Linux. Questi strumenti, ben documentati e affidabili, consentono di pianificare il lancio di applicazioni in momenti specifici. Se volete saperne di più in proposito, potete trovare l'indicazione di vari tutorial all'indirizzo <http://nostarch.com/automatestuff/>.

L'uso del programma di pianificazione incorporato nel vostro sistema operativo vi risparmia di dover scrivere del codice che controlli l'orologio per pianificare i vostri programmi. Se però avete solo bisogno di mettere brevemente in pausa il vostro programma, usate la funzione `time.sleep()`. Oppure, anziché usare il programma di pianificazione del vostro sistema operativo, il vostro codice può iterare fino a una certa data e ora, chiamando `time.sleep(1)` a ogni passaggio nel ciclo.

## Aprire siti web con Python

La funzione `webbrowser.open()` può lanciare un browser web dal vostro programma in modo che si apra su un sito specifico, anziché aprire l'applicazione browser con `subprocess.Popen()`. Vedete “Progetto: mapIt.py con il modulo `webbroser`” a pagina 254 per maggiori particolari.

## Eseguire altri script Python

Potete lanciare da Python uno script Python come qualsiasi altra applicazione. Dovete solo passare a `Popen()` l'eseguibile `python.exe` e il nome di file dello script `.py` che volete eseguire come argomento. Per esempio, quanto segue manderebbe in esecuzione lo script `hello.py` del [Capitolo 1](#):

```
>>> subprocess.Popen(['C:\\python35-32\\python.exe', 'hello.py'])  
<subprocess.Popen object at 0x00000000331CF28>
```

Passate a `Popen()` una lista che contiene una stringa con il percorso dell'eseguibile di Python e una stringa con il nome di file dello script. Se lo script che lanciate richiede argomenti dalla riga di comando, li aggiungete alla lista dopo il nome di file dello script.

La posizione dell'eseguibile di Python (di norma) in Windows è `C:\python35-32\python.exe`; in OS X è `/Library/Frameworks/Python.framework/Versions/3.3/bin/python3` e in Linux è `/usr/bin/python3`.

A differenza di quel che accade quando si importa il programma Python come un modulo, quando il vostro programma Python lancia un altro programma Python, i due vengono eseguiti in processi separate e non possono condividere l'uno le variabili dell'altro.

## Aprire file con applicazioni predefinite

Facendo un doppio clic su un file .txt sul vostro computer verrà automaticamente lanciata l'applicazione associata all'estensione di file .txt. Il vostro computer avrà già impostate parecchie di queste **associazioni per le estensioni** di file e anche Python può aprire file in questo modo con `Popen()`.

Ogni sistema operativo ha un programma che svolge l'equivalente di un doppio clic sul file di un documento per aprirlo.

In Windows, è il programma `start`; in OS X, il programma `open`; in Ubuntu Linux, il programma `see`. Inserite quanto segue nella shell interattiva, passando '`start`', '`open`' o '`see`' a `Popen()`, a seconda del vostro sistema:

```
>>> fileObj = open('hello.txt', 'w')
>>> fileObj.write('Hello world!')
12
>>> fileObj.close()
>>> import subprocess
>>> subprocess.Popen(['start', 'hello.txt'], shell=True)
```

Qui scriviamo Hello world! In un nuovo file `hello.txt`. Poi chiamiamo `Popen()`, passandole una lista contenente il nome del programma (nell'esempio, '`start`' per Windows) e il nome del file. Passiamo anche l'argomento per parola chiave `shell=True`, che è necessario solo in Windows. Il sistema operativo conosce tutte le associazioni dei tipi di file e può immaginare di dover lanciare, poniamo, `Notepad.exe` (Blocco note) per gestire il file `hello.txt`.

In OS X, viene usato il programma `open` per aprire sia i file di documento sia i programmi. Se avete un Mac, inserite quanto segue nella shell interattiva:

```
>>> subprocess.Popen(['open', '/Applications/Calculator.app'])
<subprocess.Popen object at 0x10202ff98>
```

Dovrebbe aprirsi l'applicazione Calcolatrice.

## La filosofia Unix

I programmi ben progettati per essere lanciati da altri programmi diventano più potenti di quel che sia il loro codice preso isolatamente. La **filosofia Unix** è un insieme di **principi di progettazione software** definiti dai programmatori del sistema operativo Unix (sulla cui base sono costruiti i moderni Linux e OS X). Dice che è meglio **scrivere programmi piccoli** e di finalità limitate ma che sono **in grado di interopere**, anziché applicazioni molto grandi e ricche di funzionalità. I programmi più piccoli sono più facili da comprendere e, se sono interoperabili, possono essere i blocchi da costruzione di applicazioni molto più potenti.

Anche le **app per gli smartphone** seguono la stessa impostazione. Se la vostra app per la ricerca di ristoranti deve visualizzare le indicazioni per arrivare a un certo locale, gli sviluppatori non hanno reinventato la ruota scrivendo il proprio codice di mappe; l'applicazione semplicemente lancia una app di mappe e le passa l'indirizzo del ristorante, così come il vostro codice Python

chiamerebbe una funzione passandole degli argomenti.

I **programmi Python** che avete scritto seguendo questo libro applicano fondamentalmente la filosofia Unix, in particolare per un aspetto importante: usano **argomenti da riga di comando** anziché le chiamate alla funzione `input()`. Se tutte le informazioni di cui ha bisogno il vostro programma possono essere fornite inizialmente, è preferibile avere quelle informazioni passate come argomenti da riga di comando, anziché aspettare che l'utente le scriva. In questo modo gli argomenti da riga di comando possono essere inseriti da un utente umano o forniti da un altro programma. Questo metodo dell'**interoperabilità** renderà i vostri programmi riutilizzabili come componenti di un altro programma.

L'unica eccezione è che non si vogliono passare come argomenti da riga di comando le password, poiché la riga di comando le può registrare, attraverso la sua funzione di cronologia dei comandi. Il vostro programma deve invece chiamare la funzione `input()` se ha bisogno che inseriate una password.

Potete scoprire di più sulla filosofia Unix consultando la pagina [http://en.wikipedia.org/wiki/Unix\\_philosophy/](http://en.wikipedia.org/wiki/Unix_philosophy/).

## Progetto: un semplice programma per il conteggio alla rovescia

Come è difficile trovare una applicazione semplice che funga da cronometro, può essere difficile trovare un'applicazione che esegua un semplice conteggio alla rovescia. Proviamo a scrivere un programma di questo genere, che attivi una sveglia alla fine del conteggio.

Ad alto livello, ecco che cosa farà il programma:

- contare alla rovescia da 60;
- eseguire un file di suoni (*alarm.wav*) quando il conteggio arriva a zero.

Questo significa che il vostro codice dovrà fare queste cose:

- entrare in pausa per un secondo fra la visualizzazione di un numero e l'altro nel conteggio alla rovescia, chiamando `time.sleep()`;
- chiamare `subprocess.Popen()` per aprire il file di suoni con l'applicazione predefinita.

Aprite una nuova finestra di file editor e salvate il file con il nome di *countdown.py*.

## Passo 1: conteggio alla rovescia

```
#! python3
# countdown.py - Un semplice conteggio alla rovescia.

import time, subprocess

❶ timeLeft = 60
while timeLeft > 0:
❷     print(timeLeft, end=' ')
❸     time.sleep(1)
❹     timeLeft = timeLeft - 1

# DA FARE: Al termine del conteggio alla rovescia, esegui un file di suoni.
```

Dopo aver importato `time` e `subprocess`, creiamo una variabile `timeLeft` in cui memorizzare il numero dei secondi rimanenti nel conteggio alla rovescia ❶. Può iniziare da 60, oppure potete modificare il valore in base alle vostre esigenze, oppure potete predisporre perché il programma riceva il valore come argomento da riga di comando.

In un ciclo `while`, visualizziamo il numero dei secondi rimanenti ❷, mettiamo in pausa per un secondo

3, poi diminuiamo di una unità la variabile `timeLeft` 4, prima che il ciclo ricominci. Il ciclo continua finché `timeLeft` rimane maggiore di 0, poi il conteggio sarà finito.

## Passo 2: eseguire il file di suoni

Esistono moduli di terze parti per eseguire file di suoni di vari formati, ma la cosa più rapida e facile è semplicemente lanciare l'applicazione che l'utente già usa per eseguire i file sonori. Il **sistema operativo** stabilirà, partendo dall'estensione di file `.wav`, quale applicazione debba lanciare per eseguire il file. Anziché `.wav`, il file potrebbe essere in qualche altro formato audio, come `.mp3` o `.ogg`.

Potete usare qualsiasi file audio che si trovi sul vostro computer alla fine del conteggio, oppure potete scaricare `alarm.wav` da <http://nostarch.com/automatestuff/>.

Aggiungete quanto segue al vostro codice:

```
#! python3
# countdown.py – Un semplice conteggio alla rovescia.

import time, subprocess

--righe omesse--

# Al termine del conteggio, esegue un file di suoni.
subprocess.Popen(['start','alarm.wav'], shell=True)
```

Al termine del ciclo `while`, verrà eseguito `alarm.wav` (o qualsiasi altro file audio di vostra scelta) per notificare all'utente che il conteggio alla rovescia è arrivato al termine. In Windows, non dimenticate di includere 'start' nella lista che passate a `Popen()`, né di passare l'argomento per parola chiave `shell=True`. In OS X, passate 'open' invece di 'start' ed eliminate `shell=True`.

Anziché eseguire un file audio, potreste salvare da qualche parte un file di testo come un messaggio del tipo *L'intervallo è finito!* e usare `Popen()` per aprirlo al termine del conteggio alla rovescia. In questo modo creerete una finestra a scomparsa con un messaggio. Oppure potreste usare la funzione `webbrowser.open()` per aprire un sito web particolare alla fine del conteggio. A differenza di alcune applicazioni di conteggio alla rovescia gratuite che potete trovare online, la sveglia del vostro programma può essere qualsiasi cosa vogliate.

## Idee per programmi simili

Un conteggio alla rovescia è semplicemente un ritardo prima di continuare l'esecuzione del programma. La stessa idea può essere utilizzata per altre applicazioni e altre funzionalità, come le seguenti:

- usare `time.sleep()` per dare all'utente la possibilità di premere Ctrl-C per annullare un'azione, per esempio l'eliminazione di file. Il programma può stampare un "Premi Ctrl-C per annullare" e poi gestire le eventuali eccezioni `KeyboardInterrupt` con enunciati `try` ed `except`;
- per un conteggio alla rovescia di lungo termine, potete usare oggetti `timedelta` per misurare il numero di giorni, ore, minuti e secondi, fino a un certo istante futuro (un compleanno, magari, o un anniversario).

## Riepilogo

L'**epoca Unix** (1 gennaio 1970 a mezzanotte, UTC) è un **riferimento standard** per molti linguaggi di programmazione, fra cui anche Python. Mentre la funzione `time.time()` del modulo `time` restituisce un **timestamp** (cioè un valore in virgola mobile che è il numero dei secondi trascorsi dall'epoca Unix), il modulo `datetime` è più adatto per eseguire **operazioni aritmetiche sulle date** e per formattare o analizzare stringhe con informazioni di data e ora.

La funzione `time.sleep()` rimarrà bloccata (cioè non ritornerà) per un certo numero di secondi. Può essere utilizzata per inserire **pause** nel vostro programma. Se invece volette pianificare l'avvio dei vostri programmi in un certo momento, le istruzioni che trovate all'indirizzo <http://nostarch.com/automatestaff/> possono dirvi come usare il programma di **pianificazione** già messo a disposizione dal vostro sistema operativo.

Il modulo `threading` viene usato per creare **più thread**, cosa utile se si devono scaricare molti file o si vogliono svolgere più attività simultaneamente. Fate attenzione a che il thread legga e scriva **solo variabili locali**, altrimenti potreste incorrere in problemi di concorrenza.

Infine, i vostri programmi Python possono **lanciare altre applicazioni** con la funzione `subprocess.Popen()`. Si possono passare alla chiamata `Popen()` argomenti da riga di comando per aprire specifici documenti con l'applicazione. In alternativa, potete usare con `Popen` il programma `start`, `open` o `see` per utilizzare le associazioni predefinite del vostro computer per stabilire automaticamente quale applicazione usare per l'apertura di un documento. Utilizzando le altre applicazioni presenti sul vostro computer, i vostri programmi Python possono sfruttarne le capacità per le vostre esigenze di automazione.

## Domande di ripasso

1. Che cos'è l'epoca Unix?
2. Quale funzione restituisce il numero di secondi passati dall'epoca Unix?
3. Come si può mettere in pausa un programma esattamente per 5 secondi?
4. Che cosa restituisce la funzione `round()`?
5. Qual è la differenza fra un oggetto `datetime` e un oggetto `timedelta`?
6. Supponiamo abbiate una funzione che si chiama `spam()`. Come potete chiamare questa funzione ed eseguire il codice al suo interno in un thread separato?
7. Che cosa dovete fare per evitare problemi di concorrenza con thread multipli?
8. Come potete fare in modo che un vostro programma Python esegua il programma `calc.exe` che si trova nella cartella `C:\Windows\System32`?

## Un po' di pratica

Per esercitarvi, scrivete programmi che svolgano le attività seguenti.

## Un cronometro più elegante

Ampliate il progetto del cronometro visto in questo capitolo utilizzando i metodi `rjust()` e `ljust()` per rendere “più elegante” l'output. (Abbiamo esaminato questi metodi nel [Capitolo 6](#).) Anziché presentarsi in questo modo:

Tappa #2: 8.63 (5.07)  
Tappa #3: 17.68 (9.05)  
Tappa #4: 19.11 (1.43)

L'output dovrà presentarsi in questo modo:

Tappa # 1: 3.56 ( 3.56)  
Tappa # 2: 8.63 ( 5.07)  
Tappa # 3: 17.68 ( 9.05)  
Tappa # 4: 19.11 ( 1.43)

Notate che avrete bisogno di versioni stringa delle variabili intere e in virgola mobile `lapNum`, `lapTime` e `totalTime` per chiamare i metodi stringa su di esse.

Poi, usate il modulo `pyperclip` introdotto nel [Capitolo 5](#) per copiare l'output testuale negli Appunti, in modo che l'utente possa rapidamente incollarlo in un file di testo o in un messaggio di posta elettronica.

## Pianificazione dello scaricamento di fumetti dal Web

Scrivete un programma che controlli vari siti web di fumetti e scarichi automaticamente le immagini se il sito è stato aggiornato dopo l'ultima visita del programma. Il pianificatore del sistema operativo (Utilità di pianificazione in Windows, `launchd` in OS X, `cron` in Linux) può eseguire il vostro programma Python una volta al giorno. Il programma Python a sua volta può scaricare le strisce e copiarle sul desktop in modo che sia facile trovarle. Questo vi libererà dal dover controllare il sito web personalmente per vedere se presenta qualcosa di nuovo. (Trovate un elenco di siti web di fumetti su <http://nostarch.com/automatestuff/>.)

# Inviare email e sms

Controllare e rispondere ai messaggi di posta elettronica è un compito che richiede una grande quantità di tempo. Ovviamente non potete scrivere un programma che gestisca tutta la posta per voi, perché ogni messaggio richiede una risposta appropriata, ma potete comunque **automatizzare** molte **attività legate alla posta**, se sapete come scrivere programmi che possano inviare e ricevere email.

Per esempio, potreste avere un foglio di calcolo pieno di dati sui clienti e voler spedire a ciascun cliente una circolare diversa, in base alla loro età e al luogo in cui abitano. Un software commerciale potrebbe non essere in grado di fare una cosa del genere per voi, ma per fortuna potrete scrivere un vostro programma per spedire quelle email, risparmiandovi tutto il tempo necessario per le operazioni di copia e incolla.

Potete anche scrivere programmi che inviano email e sms per notificarvi quello che accade mentre siete lontani dal vostro computer.

Se automatizzate un'attività che richiede un paio d'ore, non volette dover tornare al computer ogni pochi minuti per verificare lo status del programma. Il programma potrebbe semplicemente mandare un sms al vostro telefono quando ha finito, consentendovi di concentrarvi su altre cose più importanti mentre siete lontani dal vostro elaboratore.

## SMTP

Come HTTP è il protocollo utilizzato dai computer per inviare pagine web via Internet, **Simple Mail Transfer Protocol** (SMTP) è il **protocollo utilizzato per inviare email**. SMTP stabilisce come i messaggi debbano essere formattati, cifrati e trasferiti tra i server di posta e precisa tutti gli altri dettagli che il vostro computer deve gestire dopo che avete fatto clic su *Invia*. Non c'è bisogno però che conosciate questi dettagli tecnici, perché il modulo `smtplib` di Python li semplifica,

riconducendoli a poche funzioni.

SMTP riguarda solamente l'invio di email ad altri. Un altro protocollo, IMAP, riguarda il recupero delle email inviate a voi ed è descritto a [pagina 376](#).

## Invio di email

Probabilmente siete abituati a spedire email da Outlook o Thunderbird o attraverso siti web come Gmail o Yahoo! Mail. Purtroppo, Python non offre un'interfaccia grafica piacevole come questi servizi. Dovete invece chiamare delle funzioni per svolgere ciascun passaggio importante di SMTP, come si vede nell'esempio seguente dalla shell interattiva.

### NOTA

Non inserite questo esempio in IDLE; non funzionerà, perché `smtp.example.com`, `bob@example.com`, `MY_SECRET_PASSWORD` e `alice@example.com` sono solamente dei segnaposto. Questo codice è solo una panoramica del processo di invio di email con Python.

```
>>> import smtplib  
>>> smtpObj = smtplib.SMTP('smtp.example.com', 587)  
>>> smtpObj.ehlo()  
(250, b'mx.example.com at your service, [216.172.148.131]\nSIZE 35882577\\  
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')  
>>> smtpObj.starttls()  
(220, b'2.0.0 Ready to start TLS')  
>>> smtpObj.login('bob@example.com', 'MY_SECRET_PASSWORD')  
(235, b'2.7.0 Accepted')  
>>> smtpObj.sendmail('bob@example.com', 'alice@example.com', 'Subject: So long.\nDear Alice, so long and thanks for all  
the fish. Since rely, Bob')  
{  
>>> smtpObj.quit()  
(221, b'2.0.0 closing connection ko10sm23097611pb.52 - gsmtp')
```

Nelle sezioni seguenti, esamineremo ciascun passaggio, sostituendo i segnaposto con le informazioni giuste per connettervi e accedere a un server SMTP, inviare un messaggio di posta elettronica e disconnettervi dal server.

## Connettersi a un server SMTP

Se avete mai impostato Thunderbird, Outlook, o qualche altro programma per connettervi al vostro account di posta elettronica, saprete già che cosa significhi **configurare server SMTP e porta**. Queste impostazioni sono diverse per ciascun servizio di posta, ma una ricerca sul Web di *<vostra provider> smtp settings* dovrebbe permettervi di trovare le indicazioni che vi servono.

Il nome di dominio del server SMTP di solito è il nome di dominio del provider, con `smpt.` posto davanti. Per esempio, il server SMTP di Gmail è `smtp.gmail.com`. La [Tabella 16.1](#) elenca alcuni dei provider più diffusi e i loro server SMTP. (La porta è un valore intero e quasi sempre è 587, utilizzato dallo standard di cifratura dei comandi, **TLS**.)

**Tabella 16.1** - Provider di posta elettronica e relativi server SMTP.

Provider	Nome di dominio del server SMTP
Gmail	smtp.gmail.com
<a href="#">Outlook.com/Hotmail.com</a>	smtp-mail.outlook.com
Yahoo Mail	smtp.mail.yahoo.com
AT&T	smpt.mail.att.net (porta 465)
Comcast	smtp.comcast.net
Verizon	smtp.verizon.net (porta 465)

Una volta che disponete del nome di dominio e dell'informazione sulla porta per quanto riguarda il vostro provider, create un oggetto SMTP chiamando `smplib.SMTP()`, passando il nome di dominio come argomento stringa e la porta come argomento intero. L'oggetto SMTP rappresenta una connessione a un server di posta SMTP e ha metodi per l'invio di email. Per esempio, la chiamata seguente crea un oggetto SMTP per connettersi a Gmail:

```
>>> smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
>>> type(smtpObj)
<class 'smtplib.SMTP'>
```

L'inserimento di `type(smtpObj)` mostra che c'è un oggetto SMTP memorizzato in `smtpObj`. Questo oggetto SMTP vi servirà per chiamare i metodi che vi fanno accedere e inviare email.

Se la chiamata `smplib.SMTP()` non ha successo, può essere che il vostro server SMTP non supporti TLS sulla porta 587. In questo caso, dovete creare un oggetto SMTP utilizzando invece `smtplib.SMTP_SSL()` e la porta 465.

```
>>> smtpObj = smtplib.SMTP_SSL('smtp.gmail.com', 465)
```

## NOTA

Se vi capiterà di voler modificare dall'interno di una funzione il valore memorizzato in una variabile globale, dovete usare un enunciato `global` per quella variabile.

Se non siete collegati a Internet, Python solleverà un'eccezione `socket.gaierror: [Errno 11004] getaddrinfo failed`, o altra eccezione simile.

Per i vostri programmi, le differenze fra TLS e SSL non sono importanti. Dovete solo sapere quale standard di cifratura utilizzi il vostro server SMTP, per sapere come connettervi. In tutti gli esempi della shell interattiva che seguono, la variabile `smtpObj` conterrà un oggetto SMTP restituito dalla funzione `smtplib.SMTP()` o `smtplib.SMTP_SSL()`.

## Invio del messaggio “Hello” di SMTP

Una volta che avete l'oggetto SMTP, chiamate il suo metodo `ehlo()` per “dire hello” al server SMTP.

Questo saluto è il **primo passo in SMTP** ed è importante per stabilire una **connessione con il server**. Non vi serve conoscere le specifiche di questi protocolli. Ricordate solo di chiamare il metodo `ehlo()` come prima cosa dopo aver recuperato l’oggetto SMTP, altrimenti le successive chiamate ai metodi daranno errori. Quello che segue è un esempio di una chiamata `ehlo()` e del valore che ritorna:

```
>>> smtpObj.ehlo()
(250, b'mx.google.com at your service, [216.172.148.131]\nSIZE 35882577\
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')
```

Se il primo elemento nella tupla restituita è l’intero 250 (il codice per “andato a buon fine” in SMTP), il saluto ha avuto successo.

## Avvio della cifratura TLS

Se vi collegate alla porta 587 sul server SMTP (cioè usate la **cifratura TLS**), dovete poi chiamare il metodo `starttls()`. Questo è un passaggio necessario e attiva la cifratura per la connessione. Se vi connettete alla porta 465 (con SSL) la procedura crittografica è già importata e questo passaggio va saltato.

Ecco un esempio di chiamata del metodo `starttls()`:

```
>>> smtpObj.starttls()
(220, b'2.0.0 Ready to start TLS')
```

`starttls()` mette la vostra connessione SMTP in modalità TLS. Il 220 nel valore restituito vi dice che il server è pronto.

## Accesso al server SMTP

Una volta impostata la connessione cifrata al server SMTP, potete effettuare il **login** con il vostro **nome utente** (di solito è l’indirizzo di posta elettronica) e la **password**, chiamando il metodo `login()`:

```
>>> smtpObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
(235, b'2.7.0 Accepted')
```

### Le password Gmail specifiche dell’applicazione

Gmail ha un’ulteriore **funzione di sicurezza** per gli account Google, chiamata **password specifiche per l’applicazione**. Se riceve un messaggio d’errore Application-specific password required quando il vostro programma tenta di effettuare l’accesso, dovete impostare una di queste password per il vostro script Python. Consultate le risorse all’indirizzo <http://nostarch.com/automatestuff/> per indicazioni precise su come impostare una password specifica per l’applicazione per un account Google.

Passate come primo argomento una stringa con il vostro indirizzo di posta elettronica e come secondo argomento una stringa con la password. Il 235 nel valore di ritorno significa che l’autenticazione è andata a buon fine. Nel caso di password errata, Python solleverà un’eccezione `smtplib.SMTPAuthenticationError`.

Fate attenzione a inserire password nel codice sorgente. Se qualcuno mai dovesse copiare il vostro programma, avrebbe accesso al vostro account di posta elettronica! È una buona idea invece chiamare `input()` e lasciare che sia l'utente a scrivere la propria password. Può essere scomodo dover inserire una password ogni volta che eseguite il programma, ma in questo modo non lascerete la password in un file in chiaro sul vostro computer, dove un hacker o un ladro potrebbero trovarla facilmente.

## Invio di un messaggio di posta

Una volta effettuato l'accesso al server SMTP del vostro provider, potete chiamare il metodo `sendmail()` per spedire effettivamente il messaggio di posta. La chiamata del metodo `sendmail()` è fatta in questo modo:

```
>>> smtpObj.sendmail('my_email_address@gmail.com', 'recipient@example.com',
'Subject: A presto.\nCara Alice, a presto e grazie per tutto il pesce. Sinceramente tuo, Bob')
{}
```

Il metodo `sendmail()` richiede tre argomenti:

- il vostro indirizzo di posta elettronica sotto forma di stringa (per l'indirizzo “Da:” del messaggio).
- l'indirizzo email del destinatario, sotto forma di stringa o di lista di stringhe se i destinatari sono più di uno (per l'indirizzo “A:”).
- il corpo del messaggio sotto forma di stringa.

La stringa del corpo del messaggio deve iniziare con 'Subject: \n' per la riga dell'oggetto del messaggio. Il carattere '\n' separa la riga dell'oggetto dal corpo principale del messaggio.

Il valore restituito da `sendmail()` è un dizionario. Nel dizionario vi sarà una coppia chiave-valore per ciascun destinatario per cui la consegna del messaggio è fallita. Un dizionario vuoto significa che a tutti i destinatari il messaggio è stato inviato con successo.

## Disconnettersi dal server SMTP

Ricordatevi di chiamare il metodo `quit()`, quando avete finito di inviare email. Questo effettuerà la disconnessione del programma dal server SMTP.

```
>>> smtpObj.quit()
(221, b'2.0.0 closing connection ko10sm23097611pbd.52 - gsmtp')
```

Il 221 nel valore di ritorno significa che la sessione si sta chiudendo.

Per un riepilogo di tutti i passaggi per connettervi e accedere al server, inviare messaggi e disconnettervi, vedete "[Invio di email](#)" a pagina 372.

## IMAP

Come SMTP è il protocollo per l'invio di messaggi, **Internet Message Access Protocol (IMAP)**

specificare come comunicare con il server di un provider per recuperare i messaggi inviati al vostro indirizzo di posta elettronica. Python è fornito di un modulo `imaplib`, ma in realtà un modulo di terze parti, `imapclient`, è più facile da usare. Questo capitolo presenta un'introduzione all'uso di `IMAPClient`; la documentazione completa si può trovare all'indirizzo <http://imapclient.readthedocs.org/>. Installate `imapclient` e `pyzmail` da una finestra di Terminale. Nell'Appendice A trovate le istruzioni su come installare moduli di terze parti.

## Recuperare ed eliminare messaggi con IMAP

Trovare e recuperare un messaggio di posta elettronica in Python è un processo in più passi che richiede i due moduli di terze parti `imapclient` e `pyzmail`. Per darvi una panoramica, ecco un esempio completo di un accesso a un server IMAP, con la ricerca di email, il loro recupero e poi l'estrazione del testo dei messaggi.

```
>>> import imapclient
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)
>>> imapObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
'my_email_address@gmail.com' Jane Doe authenticated (Success)
>>> imapObj.select_folder('INBOX', readonly=True)
>>> UIDs = imapObj.search(['SINCE 05-Jul-2014'])
>>> UIDs
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]
>>> rawMessages = imapObj.fetch([40041], ['BODY[]', 'FLAGS'])
>>> import pyzmail
>>> message = pyzmail.PyzMessage.factory(rawMessages[40041]['BODY[]'])
>>> message.get_subject()
'Hello!'
>>> message.get_addresses('from')
[('Edward Snowden', 'esnowden@nsa.gov')]
>>> message.get_addresses('to')
[('Jane Doe', 'jdoe@example.com')]
>>> message.get_addresses('cc')
[]
>>> message.get_addresses('bcc')
[]
>>> message.text_part != None
True
>>> message.text_part.get_payload().decode(message.text_part.charset)
'Follow the money.\r\n\r\n-Ed\r\n'
>>> message.html_part != None
True
>>> message.html_part.get_payload().decode(message.html_part.charset)
'<div dir="ltr"><div>A presto, e grazie per tutto il pesce!<br><br></div>-Al<br></div>\r\n'
>>> imapObj.logout()
```

Non dovete memorizzare questi passaggi. Dopo che li avremo esaminati uno per uno, potete sempre tornare a questa panoramica per rinfrescarvi la memoria.

## Connessione a un server IMAP

Come serve un oggetto SMTP per connettersi a un server SMTP e spedire messaggi, dovete avere un

oggetto IMAPClient per connettervi a un server IMAP e **ricevere messaggi di posta**. Innanzitutto dovete conoscere il nome di dominio del server IMAP del provider, che sarà diverso dal nome di dominio del server SMTP. La [Tabella 16.2](#) elenca i server IMAP di vari provider fra i più comuni.

**Tabella 16.2** - Provider di posta elettronica e relativi server IMAP.

Provider	Nome di dominio del server IMAP
Gmail	imap.gmail.com
Outlook.com/Hotmail.com	imap-mail.outlook.com
Yahoo Mail	imap.mail.yahoo.com
AT&T	imap.mail.att.net
Comcast	imap.comcast.net
Verizon	incoming.verizon.net

Quando avete il nome di dominio del server IMAP, chiamate la funzione `imapclient_IMAPClient()` per creare un oggetto IMAPClient. La maggior parte dei provider richiede la crittografia SSL, perciò passate l'argomento `ssl=True`.

Inserite quanto segue nella shell interattiva (sostituendo a `imap.gmail.com` il nome di dominio del vostro provider):

```
>>> import imapclient  
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)
```

In tutti gli esempi della shell interattiva delle prossime pagine, la variabile `imapObj` conterrà un oggetto IMAPClient restituito dalla funzione `imapclient_IMAPClient()`. In questo contesto, un client è l'oggetto che si connette al server.

## Accesso al server IMAP

Quando avete un oggetto IMAPClient, chiamate il suo metodo `login()`, passando il nome utente (di solito è il vostro indirizzo di posta elettronica) e la password come stringhe.

```
>>> imapObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')  
'my_email_address@gmail.com Jane Doe authenticated (Success)'
```

### NOTA

Ricordate: non inserite mai una password direttamente nel vostro codice! Predisponete invece il vostro programma in modo che accetti la password restituita da `input()`.

Se il server IMAP respinge la combinazione nome utente/Password, Python solleva un'eccezione `imaplib.error`. Per gli account Gmail, probabilmente dovrete utilizzare una password specifica per l'applicazione: per saperne di più, consultate "[Le password Gmail specifiche per l'applicazione](#)" a

## Ricerca dei messaggi

Una volta effettuato l'accesso, l'effettivo recupero dei messaggi di posta a cui siete interessati è un procedimento in due passi. Innanzitutto, dovete selezionare una cartella in cui cercare; poi dovete chiamare il metodo `search()` dell'oggetto `IMAPClient`, passandogli una stringa di parole chiave di ricerca IMAP.

### Selezionare una cartella

Quasi tutti gli account hanno per default una cartella `INBOX`, ma potete ottenere una lista delle cartelle chiamando il metodo `list_folders()` dell'oggetto `IMAPClient`. Questo restituisce una lista di tuple, ciascuna delle quali contiene informazioni su una singola cartella. Continuate l'esempio nella shell interattiva inserendo quanto segue:

```
>>> import pprint
>>> pprint.pprint(imapObj.list_folders())
[('\"HasNoChildren', '/', 'Drafts'),
 ('\"HasNoChildren', '/', 'Filler'),
 ('\"HasNoChildren', '/', 'INBOX'),
 ('\"HasNoChildren', '/', 'Sent'),
--righe omesse--
('\"HasNoChildren', '\"Flagged'), '/', '[Gmail]/Starred'),
('\"HasNoChildren', '\"Trash'), '/', '[Gmail]/Trash')]
```

Questo è il probabile output se avete un account con Gmail. (Gmail chiama le sue cartelle *labels*, ma funzionano come cartelle.) I tre valori di ciascuna delle tuple, per esempio `('\"HasNoChildren', '/', 'INBOX')`, sono:

- una tupla dei flag della cartella (che cosa rappresentino esattamente questi flag va oltre le finalità di questo libro, e potete tranquillamente ignorare questo campo);
- il delimitatore utilizzato nella stringa del nome per separare le cartelle genitrici e le sottocartelle;
- il nome completo della cartella.

Per selezionare una cartella in cui effettuare una ricerca, passate il nome della cartella come stringa al metodo `select_folder()` dell'oggetto `IMAPClient`.

```
>>> imapObj.select_folder('INBOX', readonly=True)
```

Potete ignorare il valore di ritorno di `select_folder()`. Se la cartella selezionata non esiste, Python solleva un'eccezione `imaplib.error`.

L'argomento per parola chiave `readonly=True` vi impedisce di modificare o cancellare accidentalmente qualche messaggio presente nella cartella con le chiamate successive ai metodi. A meno che non vogliate espressamente cancellare dei messaggi, è una buona idea impostare sempre `readonly` a `True`.

### Eseguire la ricerca

Selezionata una cartella, potete cercare fra i messaggi con il metodo `search()` dell'oggetto `IMAPClient`. L'argomento di `search()` è una lista di stringhe, ciascuna formattata in base alle chiavi di ricerca di IMAP, descritte nella [Tabella 16.3](#).

**Tabella 16.3** - Le chiavi di ricerca IMAP.

Chiave di ricerca	Significato
'ALL'	Restituisce tutti i messaggi presenti nella cartella. Potreste incappare nei limiti di dimensione di <code>imaplib</code> , se richiedete tutti i messaggi presenti in una cartella di grandi dimensioni. Vedete il paragrafo " <a href="#">Limiti dimensionali</a> " a <a href="#">pagina 382</a> .
'BEFORE date', 'ON date', 'SINCE date'	Queste tre chiavi di ricerca restituiscono, rispettivamente, i messaggi ricevuti dal server IMAP prima, alla, o dopo la data indicata. La data deve essere formattata come 05-Jul-2017. Inoltre, mentre 'SINCE 05-Jul-2017' ritroverà i messaggi del 5 luglio e successivi, 'BEFORE 05-Jul-2015' troverà solo i messaggi con data precedente il 5 luglio, e non quelli del 5 luglio stesso.
'SUBJECT string', 'BODY string', 'TEXT string'	Restituiscono i messaggi che hanno la stringa rispettivamente nell'oggetto, nel corpo o in entrambi. Se stringa contiene degli spazi, dovete racchiuderla fra doppi apici: 'TEXT "ricerca contenente spazi"'.
'FROM string', 'TO string', 'CC string', 'BCC string'	Restituiscono tutti i messaggi in cui la stringa si trova nell'indirizzo del mittente, nell'indirizzo del destinatario, negli indirizzi CC o BCC, rispettivamente. Se <i>string</i> contiene più indirizzi email, bisogna separarli con spazi e racchiuderli fra apici doppi: 'CC "primocc@example.com secondocc@example.com"'.
'SEEN', 'UNSEEN'	Restituiscono tutti i messaggi, rispettivamente con e senza il flag \Seen ("già letto"). Un messaggio ha il flag \Seen se vi si è già acceduto con una chiamata al metodo <code>fetch()</code> (descritto più avanti) o se vi si fa clic sopra quando si controlla la posta in un programma apposito o in un browser web. Di solito si dice che si "leggono" i messaggi, non che si "vedono", ma il significato è lo stesso.
'ANSWERED', 'UNANSWERED'	Restituiscono tutti i messaggi, rispettivamente con o senza il flag \Answered. Un messaggio prende il flag \Answered quando vi si risponde.
'DELETED', 'UNDELETED'	Restituiscono tutti i messaggi rispettivamente con o senza il flag \Deleted. Ai messaggi eliminati con il metodo <code>delete_messages()</code> viene attribuito il flag \Deleted, ma non vengono cancellati definitivamente fino a che non viene chiamato il metodo <code>expunge()</code> (vedete il paragrafo " <a href="#">Cancellazione di messaggi</a> " a <a href="#">pagina 386</a> ). Notate che alcuni provider, come Gmail, eliminano automaticamente i messaggi.
'DRAFT', 'UNDRAFT'	Restituiscono tutti i messaggi, rispettivamente con o senza il flag \Draft (Bozza). I messaggi in bozza vengono normalmente conservati in una cartella distinta dalla INBOX.
'FLAGGED', 'UNFLAGGED'	Restituiscono tutti i messaggi, rispettivamente con o senza il flag \Flagged. Questo flag di solito è usato per evidenziare i messaggi "importanti" o "urgenti".
'LARGER N', 'SMALLER N'	Restituiscono tutti i messaggi le cui dimensioni sono, rispettivamente, maggiori o minori di <i>N</i> byte.
'NOT chiave-di-ricerca'	Restituisce tutti i messaggi che la <i>chiave-di-ricerca</i> non avrebbe restituito.
'OR chiave-di-ricerca1 chiave-di-ricerca2'	Restituisce i messaggi che corrisponde o alla prima o alla seconda <i>chiave-di-ricerca</i> .

Tenete presente che qualche server IMAP può implementare in modo diverso la gestione dei flag e delle chiavi di ricerca. Dovrete sperimentare un po' nella shell interattiva per vedere esattamente come si comportano.

Potete passare più stringhe con le chiavi di ricerca IMAP nell'argomento lista del metodo `search()`. I messaggi restituiti saranno quelli che soddisfano tutte le chiavi di ricerca. Se volete che i messaggi soddisfino almeno una delle chiavi di ricerca, usate la chiave `OR`. Per le chiavi `NOT` e `OR`, devono essere seguite rispettivamente da una e due chiavi di ricerca complete.

Quelli che seguono sono alcuni esempi di chiamate al metodo `search()`, con i relativi significati.

- `imapObj.search(['ALL'])` Restituisce tutti i messaggi nella cartella selezionata.
- `imapObj.search(['ON 05-Jul-2017'])` Restituisce tutti i messaggi inviati il 5 luglio 2017.
- `imapObj.search(['SINCE 01-Jan-2017', 'BEFORE 01-Feb-2017', 'UNSEEN'])` Restituisce tutti i messaggi inviati nel gennaio 2017 e non letti. (Notate che questo significa l'1 gennaio o più tardi, l'1 febbraio escluso.)
- `imapObj.search(['SINCE 01-Jan-2017', 'FROM alice@example.com'])` Restituisce tutti i messaggi inviati da `alice@example.com` dal primo giorno del 2017.
- `imapObj.search(['SINCE 01-Jan-2017', 'NOT FROM alice@example.com'])` Restituisce tutti i messaggi inviati da tutti, tranne `alice@example.com` dal primo giorno del 2017.
- `imapObj.search(['OR FROM alice@example.com FROM bob@example.com'])` Restituisce tutti i messaggi inviati da `alice@example.com` o `bob@example.com`.
- `imapObj.search(['FROM alice@example.com', 'FROM bob@example.com'])` Esempio complicato! Questa ricerca non restituirà mai alcun messaggio, perché i messaggi devono soddisfare tutte le chiavi di ricerca. Poiché ci può essere un unico indirizzo mittente, è impossibile che un messaggio arrivi sia da `alice@example.com` sia da `bob@example.com`.

Il metodo `search()` non restituisce i messaggi stessi, ma gli ID unici (UID) dei messaggi, sotto forma di valori interi. Poi potete passare questi UID al metodo `fetch()` per ottenere il contenuto dei messaggi. Continuate l'esempio nella shell interattiva inserendo quanto segue:

```
>>> UIDs = imapObj.search(['SINCE 05-Jul-2016'])
>>> UIDs
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]
```

Qui, la lista degli ID di messaggio (per i messaggi ricevuti dal 5 luglio 2016 in poi) restituita da `search()` è memorizzata in `UIDs`. La lista degli UID restituita dal vostro computer sarà ovviamente diversa da quella indicata qui: gli ID sono specifici per i singoli account di posta. Quando poi passate gli UID ad altre chiamate di funzione, usate i valori UID ricevuti, non quelli stampati negli esempi del libro.

## ***Limiti dimensionali***

Se le vostre chiavi di ricerca sono soddisfatte da un gran numero di messaggi, Python potrebbe sollevare un'eccezione del tipo `imaplib.error: got more than 10000 bytes`. Quando succede, dovete disconnettervi, riconnettervi al server IMAP e riprovare.

Questo limite è in vigore per impedire ai programmi Python di consumare troppo spazio di memoria. Purtroppo, il limite predefinito spesso è troppo piccolo, ma potete modificarlo da 10.000 a

10.000.000 di byte con questo codice:

```
>>> import imaplib  
>>> imaplib._MAXLINE = 10000000
```

Questo dovrebbe impedire che il messaggio d'errore compaia di nuovo. Potreste magari inserire queste due righe in tutti i programmi che scrivete per IMAP.

## Il metodo `gmail_search()` di `IMAPClient`

Se vi collegate al server [imap.gmail.com](http://imap.gmail.com) per accedere a un account Gmail, l'oggetto `IMAPClient` mette a disposizione un'ulteriore funzione di ricerca che imita la barra di ricerca che si trova nella parte superiore della pagina web di Gmail, evidenziata nella Figura 16.1.



Figura 16.1 - La barra di ricerca nella pagina web di Gmail.

Anziché ricercare con le chiavi di ricerca IMAP, potete utilizzare il motore di ricerca di Gmail, che è più raffinato. Gmail svolge un ottimo lavoro, cercando anche parole strettamente legate alle chiavi di ricerca (per esempio, se si cerca *driving* compariranno anche i risultati per *drive* e *drove*) e ordina i risultati in base alla pertinenza. Potete anche utilizzare gli operatori di ricerca avanzati di Gmail (maggiori informazioni su <http://nostarch.com/automatestuff/>). Se accedete a un account di Gmail, passate i termini di ricerca al metodo `gmail_search()` anziché a `search()`, come in questo esempio:

```
>>> UIDs = imapObj.gmail_search('significato della vita')  
>>> UIDs  
[42]
```

Ah sì, eccolo lì quel messaggio con il significato della vita! Lo stavo proprio cercando.

## Recuperare un messaggio e marcarlo come già letto

Quando avete una lista di UID, potete chiamare il metodo `fetch()` dell'oggetto `IMAPClient` per ottenere i contenuti effettivi dei messaggi.

La lista degli UID sarà il primo argomento di `fetch()`; il secondo deve essere la lista `['BODY[]']`, che dice a `fetch()` di scaricare tutti i contenuti delle email specificate nella lista degli UID.

Continuiamo il nostro esempio nella shell interattiva.

```
>>> rawMessages = imapObj.fetch(UIDs, ['BODY[]'])  
>>> import pprint  
>>> pprint.pprint(rawMessages)  
{40040: {'BODY[]': 'Delivered-To: my_email_address@gmail.com\r\n'  
         'Received: by 10.76.71.167 with SMTP id '  
         '--rige omesse--  
         '\r\n'  
         '-----_Part_6000970_707736290.1404819487066--\r\n',  
         'SEQ': 5430}}
```

Importate `pprint` e passate il valore di ritorno da `fetch()`, memorizzato nella variabile `rawMessages`, a

`pprint.pprint()` per avere una stampa elegante, e vedrete che questo valore di ritorno è un dizionario annidato di messaggi con gli UID come chiavi. Ciascun messaggio è memorizzato come un dizionario con due chiavi: 'BODY[]' e 'SEQ'. La prima corrisponde al corpo effettivo del messaggio; la seconda è per un numero di sequenza, che ha un ruolo simile all'UID e che potete tranquillamente ignorare. Come potete vedere, il contenuto del messaggio nella chiave 'BODY[]' è del tutto incomprensibile. È in un formato denominato **RFC 822**, pensato per essere letto dai server IMAP. Non c'è bisogno che conosciate il formato RFC 822; nel seguito del capitolo, se ne farà carico il modulo `pymail`. Quando avete selezionato una cartella in cui cercare, avete chiamato `select_folder()` con l'argomento per parola chiave `readonly=True`. In questo modo vi tutelavate dalla possibilità di cancellare accidentalmente un messaggio, ma questo significa anche che le email non verranno marcate come lette se le scaricate con il metodo `fetch()`. Se volete che i messaggi vengano marcati come letti quando li scaricate, dovete passare a `select_folder()` l'argomento `readonly=False`. Se la cartella selezionata è già in modalità sola lettura, potete riselectare la cartella corrente con un'altra chiamata a `select_folder()`, questa volta con l'argomento per parola chiave `readonly=False`:

```
>>> imapObj.select_folder('INBOX', readonly=False)
```

## Ottenere indirizzi di posta elettronica da un messaggio grezzo

I messaggi grezzi restituiti dal metodo `fetch()` non sono molto utili per chi vuole semplicemente leggere le proprie email. Il modulo `pymail` analizza questi messaggi grezzi e li restituisce come oggetti `PyzMessage`, che rendono facilmente accessibili al codice Python l'oggetto, il corpo, i campi "Da" e "A" e altre parti dei messaggi.

Continuate l'esempio nella shell interattiva con quanto segue (utilizzando gli UID del vostro account email, non quelli indicati qui a titolo di esempio):

```
>>> import pymail  
>>> message = pymail.PyzMessage.factory(rawMessages[40041]['BODY[]'])
```

Per prima cosa, importate `pymail`. Poi, per creare un oggetto `PyzMessage` per una email, chiamate la funzione `pymail.PyzMessage.factory()` e le passate la sezione 'BODY[]' del messaggio grezzo. Memorizzate il risultato in `message`. Ora `message` contiene un oggetto `PyzMessage`, che dispone di parecchi metodi che semplificano il recupero dell'oggetto del messaggio e degli indirizzi di mittente e destinatario. Il metodo `get_subject()` restituisce l'oggetto come un semplice valore stringa. Il metodo `get_addresses()` restituisce una lista di indirizzi per i campi che si passano. Ecco un esempio:

```
>>> message.get_subject()  
'Hello!'  
>>> message.get_addresses('from')  
[['Edward Snowden', 'esnowden@nsa.gov']]  
>>> message.get_addresses('to')  
[['Jane Doe', 'my_email_address@gmail.com']]  
>>> message.get_addresses('cc')  
[]  
>>> message.get_addresses('bcc')  
[]
```

Notate che l'argomento per `get_addresses()` è 'from', 'to', 'cc' o 'bcc'. Il valore di ritorno di `get_addresses()` è una

lista di tuple. Ciascuna tupla contiene due stringhe: la prima è il nome associato all'indirizzo email, la seconda l'indirizzo stesso. Se non vi sono indirizzi nel campo richiesto, `get_addresses()` restituisce una lista vuota. Qui, i campi 'cc' e 'bcc' non contenevano alcun indirizzo, perciò sono state restituite liste vuote.

## Ottenere il corpo da un messaggio grezzo

Le email possono essere inviate in forma di **puro testo**, come **HTML** o in entrambi i modi. Nel primo caso contengono solo testo, mentre i messaggi in HTML possono avere colori, tipi di caratteri diversi, immagini e altre caratteristiche che fanno assomigliare un messaggio a una piccola pagina web. Se un messaggio è in puro testo, il suo oggetto `PyzMessage` avrà gli attributi `html_part` impostati a `None`. Analogamente, se un messaggio è di solo HTML, il suo oggetto avrà l'attributo `text_part` impostato a `None`.

Altrimenti, il valore `text_part` o `html_part` avrà un metodo `get_payload()` che restituisce il corpo del messaggio come un valore del tipo di dati `bytes`. (Il tipo di dati `bytes` va al di là delle finalità di questo libro.) Ma questo ancora non è un valore stringa che possiamo usare. L'ultimo passo sta nel chiamare il metodo `decode()` sul valore `bytes` restituito da `get_payload()`. Il metodo `decode()` prende un argomento: la codifica dei caratteri del messaggio, memorizzata nell'attributo `text_part.charset` o `html_part.charset`. Questo, finalmente, restituirà la stringa del corpo del messaggio.

Continuate l'esempio nella shell interattiva inserendo quanto segue:

```
❶ >>> message.text_part != None
True
>>> message.text_part.get_payload().decode(message.text_part.charset)
❷ 'A presto, e grazie per tutto il pesce!\r\n\r\n-AI\r\n'
❸ >>> message.html_part != None
True
❹ >>> message.html_part.get_payload().decode(message.html_part.charset)
'<div dir="ltr"><div>A presto, e grazie per tutto il pesce!<br><br></div>-AI
<br></div>\r\n'
```

Il messaggio con cui stiamo lavorando ha sia testo puro sia contenuti in HTML, perciò l'oggetto `PyzMessage` memorizzato in `message` ha gli attributi `text_part` e `html_part` diversi da `None` ❶ ❸. Chiamando `get_payload()` su `text_part` del messaggio e poi chiamando `decode()` sul valore `bytes` si ha di ritorno una stringa con la versione testuale del messaggio ❷. Con `get_payload()` e `decode()` sulla `html_part` del messaggio si ha di ritorno una stringa con la versione HTML del messaggio ❹.

## Cancellazione di messaggi

Per cancellare email, si passa una lista di UID di messaggio al metodo `delete_messages()` dell'oggetto `IMAPClient`. Questo marca le email con il flag `\Deleted`. Chiamando il metodo `expunge()` si cancelleranno definitivamente tutti i messaggi con il flag `\Deleted` nella cartella selezionata. Guardate questo esempio nella shell interattiva:

```
❶ >>> imapObj.select_folder('INBOX', readonly=False)
❷ >>> UIDs = imapObj.search(['ON 09-Jul-2015'])
❸ >>> UIDs
[40066]
>>> imapObj.delete_messages(UIDs)
❹ {40066: ('\\Seen', '\\Deleted')}
>>> imapObj.expunge()
('Success', [(5452, 'EXISTS')])
```

Qui selezioniamo la casella della posta in arrivo chiamando `select_folder()` sull'oggetto `IMAPClient` e passando '`INBOX`' come primo argomento; passiamo anche l'argomento per parola chiave `readonly=False` in modo da poter cancellare messaggi ❶. Cerchiamo nella casella i messaggi ricevuti in una data specifica e memorizziamo gli ID dei messaggi restituiti in `UIDs` ❷.

Chiamando `delete_message()` e passando `UIDs` si ha di ritorno un dizionario; ogni coppia chiave-valore è un ID di messaggio con una tupla dei flag del messaggio, che ora dovrebbe contenere anche `\Deleted` ❸.

Una chiamata a `expunge()` poi cancella definitivamente i messaggi con il flag `\Deleted` e restituisce un messaggio che tutto è andato a buon fine, se non sono stati incontrati problemi nell'eliminazione definitiva delle email.

Notate che alcuni provider, per esempio Gmail, eliminano automaticamente i messaggi cancellati con `delete_messages()`, anziché attendere il comando `expunge()` dal client IMAP.

## Disconnessione dal server IMAP

Quando il programma ha finito di recuperare o cancellare email, semplicemente chiamate il metodo `logout()` di `IMAPClient` per disconnettervi dal server IMAP.

```
>>> imapObj.logout()
```

Se il vostro programma gira per un po' di minuti, è possibile che il server IMAP invii un **time out**, ovvero che si disconnetta automaticamente. In questo caso, la chiamata successiva a un metodo da parte del programma farà sì che l'oggetto `IMAPClient` sollevi un'eccezione come questa:

```
imaplib.abort: socket error: [WinError 10054] An existing connection was
forcibly closed by the remote host
```

Se succede, il vostro programma dovrà chiamare `imapclient.IMAPClient()` per connettersi nuovamente. Questo è tutto. Ci sono stati molti ostacoli da superare, ma ora avete un modo per far sì che i vostri programmi Python accedano a un account di posta elettronica e scarichino i messaggi. Potete sempre riguardare la panoramica su "["Recuperare ed eliminare messaggi con IMAP"](#)" a pagina 376, se dovete rinfrescarvi la memoria sui vari passaggi.

## Progetto: inviare email di sollecito ai membri di un club

Supponiamo che vi siate "offerti volontari" di tener traccia del pagamento delle quote degli associati a un club di volontari. È un compito molto noioso, che comporta tenere un foglio di calcolo in cui segnare chi ha pagato nei vari mesi e inviare dei solleciti per posta elettronica a chi è in arretrato. Anziché analizzare personalmente il foglio di calcolo, e copiare e incollare la stessa email per tutti

quelli che vanno sollecitato, proviamo a realizzare uno script che lo faccia per voi. Ad alto livello, ecco che cosa dovrà fare il programma:

- leggere dati da un foglio di Excel;
  - identificare tutti i soci che non hanno pagato la loro quota nell'ultimo mese;
  - trovare i loro indirizzi di posta elettronica e inviare solleciti personalizzati.

Questo significa che il vostro codice dovrà fare queste cose:

- aprire e leggere le celle di un documento Excel con il modulo `openpyxl` (lo abbiamo analizzato nel [Capitolo 12](#));
  - creare un dizionario degli associati in arretrato con le loro quote;
  - collegarsi a un server SMTP chiamando `smtplib.SMTP()`, `ehlo()`, `starttls()` e `login()`;
  - per tutti gli associati in arretrato, inviare un sollecito personalizzato per posta elettronica, chiamando il metodo `sendmail()`.

Aprite una nuova finestra di file editor e salvate con il nome *sendDuesReminders.py*.

## Passo 1: aprire il file Excel

Supponiamo che il foglio Excel che utilizzate per tener traccia del pagamento delle quote sia come quello della Figura 16.2 e che si trovi in un file chiamato *duesRecords.xlsx*. Potete scaricare questo file da <http://nostarch.com/automatestuff/>.

**Figura 16.2** - Il foglio in cui si tiene traccia delle quote pagate

Questo foglio contiene il nome e l'indirizzo email dei vari soci. Per ogni mese c'è una colonna che tiene traccia di chi ha pagato o meno.

Il programma dovrà aprire *duesRecord.xlsx* e stabilire qual è la colonna dell'ultimo mese chiamando il metodo `get_highest_column()`. (Potete consultare il [Capitolo 12](#) per ulteriori informazioni su come accedere alle celle nei file Excel con il modulo `openpyxl`.) Inserite il codice seguente nella finestra di

file editor:

```
#! python3
# sendDuesReminders.py - Invia email in base allo stato dei pagamenti in un foglio Excel.

import openpyxl, smtplib, sys

❶ #     Apre il foglio e ricava lo stato delle quote.
❷ wb = openpyxl.load_workbook('duesRecords.xlsx')
❸ sheet = wb.get_sheet_by_name('Sheet1')

❹ lastCol = sheet.get_highest_column()
❺ latestMonth = sheet.cell(row=1, column=lastCol).value

# DA FARE: Verificare lo stato dei pagamenti di ciascuno.

# DA FARE: Accedere all'account di posta.

# DA FARE: Inviare email di sollecito.
```

Importati i moduli `openpyxl`, `smtplib` e `sys`, apriamo il file `duesRecords.xlsx` e memorizziamo l’oggetto `Workbook` risultante in `wb` ❶. Poi andiamo al primo foglio, Sheet 1, e memorizziamo l’oggetto `Worksheet` risultante in `sheet` ❷. Ora che abbiamo un oggetto `Worksheet`, possiamo accedere alle righe, alle colonne e alle celle. Memorizziamo l’ultima colonna in `lastCol` ❸, poi usiamo la riga numero 1 e `lastCol` per accedere alla cella che deve indicare l’ultimo mese. Otteniamo il valore che si trova in questa cella e lo memorizziamo in `latestMonth` ❹.

## Passo 2: trovare tutti gli associati che non hanno pagato

Determinato il numero di colonna dell’ultimo mese (memorizzato in `lastCol`), possiamo iterare su tutte le righe dopo la prima (che contiene le intestazioni di colonna), per vedere per quali membri la cella dei pagamenti di quel mese contiene il testo `paid`. Se quella persona non ha pagato la sua quota, possiamo prendere nome e indirizzo email dalle colonne 1 e 2, rispettivamente. Queste informazioni vanno nel dizionario `unpaidMembers`, che terrà traccia di tutti gli associati che non hanno pagato la quota dell’ultimo mese. Aggiungete il codice seguente a `sendDuesReminder.py`.

```
#! python3
# sendDuesReminders.py - Invia email in base allo stato dei pagamenti in un foglio Excel.

--righe omesse--

# Controlla lo stato dei pagamenti di ciascun associato.
unpaidMembers = {}
❶ for r in range(2, sheet.get_highest_row() + 1):
❷     payment = sheet.cell(row=r, column=lastCol).value
❸     if payment != 'paid':
❹         name = sheet.cell(row=r, column=1).value
❺         email = sheet.cell(row=r, column=2).value
❻         unpaidMembers[name] = email
```

Questo codice imposta un dizionario `unpaidMembers` vuoto e poi itera su tutte le righe dopo la prima ❶. Per ciascuna riga, il valore della colonna più recente è memorizzato in `payment` ❷. Se `payment` non è

uguale a 'paid', allora il valore della prima colonna viene memorizzato in name ❸, il valore della seconda colonna è memorizzato in email ❹ e name ed email vengono aggiunti a unpaidMembers ❺.

## Passo 3: inviare solleciti personalizzati per posta elettronica

Disponendo della lista di tutti gli associati che non hanno pagato la loro quota, è venuto il momento di inviare loro un sollecito per posta elettronica. Aggiungete al vostro programma il codice seguente, ma sostituite i segnaposto con il vostro indirizzo di posta elettronica e le informazioni sul vostro provider.

```
#! python3
# sendDuesReminders.py – Invia email in base allo stato dei pagamenti in un foglio Excel.

--righe omesse--
```

```
# Accede all'account di posta elettronica.
smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
smtpObj.ehlo()
smtpObj.starttls()
smtpObj.login('my_email_address@gmail.com', sys.argv[1])
```

Create un oggetto SMTP chiamando `smtplib.SMTP()` e passando il nome di dominio e la porta del vostro provider. Chiamate `ehlo()` e `starttls()`, poi `login()` e passate il vostro indirizzo email e `sys.argv[1]`, che memorizzerà la stringa della vostra password. Inserirete la password come argomento da riga di comando ogni volta che eseguirete il programma, per non salvarla nel codice sorgente.

Quando il programma ha ottenuto l'accesso all'account di posta, deve percorrere il dizionario `unpaidMembers` e inviare un messaggio personalizzato all'indirizzo email di ciascun socio. Aggiungete quanto segue a `sendDuesReminders.py`:

```
#! python3
# sendDuesReminders.py – Invia email in base allo stato dei pagamenti in un foglio Excel.

--righe omesse--
```

```
# Invia email di sollecito.
for name, email in unpaidMembers.items():
❶    body = "Subject: %s Quote non versate.\nGentile %s,\nDalla nostra documentazione
    risulta che non ha ancora versato la quota per %s. Per favore effettui il pagamento il
    più presto possibile. Grazie!" %
    (latestMonth, name, latestMonth)
❷    print('Sto mandando un'email a %s...' % email)
❸    sendmailStatus = smtpObj.sendmail('my_email_address@gmail.com', email, body)

❹    if sendmailStatus != {}:
        print('C\'è stato un problema nell\'invio del messaggio a %s: %s' % (email,
        sendmailStatus))
    smtpObj.quit()
```

Questo codice cicla su nomi e indirizzi email in `unpaidMember`. Per ciascun associato che non ha versato la sua quota, personalizziamo un messaggio con l'ultimo mese e il nome della persona e memorizziamo il messaggio in `body` ❶. Stampiamo un output che dice che il programma sta inviando

un messaggio all'indirizzo email di quella persona ❷. Poi chiamiamo `sendmail()`, passando l'indirizzo "Da:" e il messaggio personalizzato ❸. Il valore restituito viene memorizzato in `sendmailStatus`. Ricordate che il metodo `sendmail()` restituisce un valore di dizionario non vuoto se il server SMTP riferisce un errore nell'invio di quella particolare email. L'ultima parte del ciclo `for` in ❹ controlla se il dizionario restituito non è vuoto ed eventualmente stampa l'indirizzo email del destinatario che non è stato raggiunto e il dizionario restituito. Quando il programma ha finito di inviare tutte le email, viene chiamato il metodo `quit()` per la disconnessione dal server SMTP.

Quando si esegue il programma, l'output sarà qualcosa di analogo a questo:

```
Sto mandando un'email a alice@example.com...
Sto mandando un'email a bob@example.com...
Sto mandando un'email a eve@example.com...
```

I destinatari riceveranno un messaggio di posta elettronica come quello della Figura 16.3.



Figura 16.3 - Un messaggio di posta elettronica inviato automaticamente da `sendDuesReminders.py`.

## Inviare sms con Twilio

È molto più probabile trovare una persona vicino al suo telefono che al suo computer, perciò gli sms possono costituire un modo più immediato e affidabile delle email per inviare notifiche. Inoltre, la brevità degli sms rende molto più probabile che chi li riceve si dia la pena di leggerli.

In questa sezione, vedrete come accedere a **Twilio**, un servizio gratuito, e usare il suo modulo Python per inviare sms. Twilio è un **servizio gateway SMS**, cioè è un servizio che consente l'invio di sms dai programmi. Avrete dei limiti per quanto riguarda il numero di sms che potete spedire ogni mese e ogni sms avrà in testa le parole *Sent from a Twilio trial account*, ma questo servizio è probabilmente più che sufficiente per i vostri programmi personali.

La versione "di prova" gratuita non ha scadenza: non sarete costretti a passare a un abbonamento a pagamento più avanti.

Twilio non è l'unico gateway SMS; se preferite non usarlo, potete trovare servizi alternativi cercando online *free sms gateway*, *python sms api* o anche *twilio alternatives*.

Prima di sottoscrivere un account Twilio, installate il modulo `twilio`. Nell'Appendice A trovate maggiori informazioni sull'installazione di moduli di terze parti.

## NOTA

Questa sezione è specifica per gli Stati Uniti. Twilio offre servizi sms per Paesi diversi dagli

Stati Uniti, ma quei servizi specifici non sono trattati in questo libro. Il modulo `twilio` e le sue funzioni, però, si comporteranno nello stesso modo ovunque. Potete trovare maggiori informazioni sul sito <https://www.twilio.com>.

## Creare un account Twilio

Andate al sito <https://www.twilio.com/> e compilate il modulo di iscrizione. Istituito un nuovo account, dovrete verificare un numero telefonico a cui volete inviare sms. (La verifica è necessaria per impedire l'uso del servizio per inviare sms spam a numeri telefonici scelti a caso.)

Dopo aver ricevuto il messaggio con il numero di verifica, inseritelo nel sito web di Twilio per dimostrare che siete i legittimi proprietari del telefono che state verificando. A quel punto sarete in grado di inviare sms a quel numero attraverso il modulo `twilio`.

Twilio fornisce all'account di prova un numero telefonico da usare come mittente degli sms. Vi serviranno altre due informazioni: il **SID dell'account** e l'**auth token** (token di autorizzazione). Potete trovare queste informazioni sulla pagina *Dashboard* dopo aver effettuato l'accesso al vostro account. Questi valori si comportano come nome utente e password di Twilio quando effettuate l'accesso da un programma Python.

## Invio di sms

Installato il modulo `twilio`, creato un account Twilio, verificato il numero di telefono, registrato un numero telefonico Twilio e ottenuti SID dell'account e auth token, siete finalmente pronti per inviarvi sms da script Python. Rispetto a tutti i passaggi per la registrazione, il codice Python è piuttosto semplice. Con il computer collegato a Internet, inserite quanto segue nella shell interattiva, sostituendo i segnaposto `accountSID`, `authToken`, `myTwilioNumber` e `myCellPhone` con le vostre informazioni reali:

```
u  >>> from twilio.rest import TwilioRestClient
>>> accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
>>> authToken = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
v  >>> twilioCli = TwilioRestClient(accountSID, authToken)
>>> myTwilioNumber = '+14955551234'
>>> myCellPhone = '+14955558888'
w  >>> message = twilioCli.messages.create(body='Mr. Watson - Venga - Voglio
vederla.', from_=myTwilioNumber, to=myCellPhone)
```

Qualche istante dopo aver scritto l'ultima riga, dovreste ricevere un messaggio di testo che dice *Sent from your Twilio trial account – Mr. Watson – Venga – Voglio vederla.*

Dato il modo in cui è impostato il modulo `twilio`, dovete importarlo con `from twilio.rest import TwilioRestClient`, non semplicemente con `import twilio` ❶. Memorizzate il SID del vostro account in `accountSID` e l'`auth token` in `authToken` e poi chiamate `TwilioRestClient()` passando `accountSID` e `authToken`. La chiamata a `TwilioRestClient()` restituisce un oggetto `TwilioRestClient` ❷. Questo oggetto ha un attributo `message`, che a sua volta ha un metodo `create()` che potete usare per inviare sms. Questo è il metodo che dice ai server di Twilio di inviare il vostro sms. Memorizzati numero di Twilio e numero del cellulare in `myTwilioNumber` e `myCellPhone`, rispettivamente, chiamate `create()` e passategli argomenti per parola chiave che specificano il corpo del messaggio, il numero del mittente (`myTwilioNumber`) e quello del destinatario (`myCellPhone`) ❸.

L'oggetto Message restituito dal metodo `create()` conterrà informazioni sul messaggio inviato. Continuate l'esempio nella shell interattiva inserendo quanto segue:

```
>>> message.to  
'+14955558888'  
>>> message.from_  
'+14955551234'  
>>> message.body  
'Mr. Watson - Venga qui - Voglio vederla.'
```

Gli attributi `to`, `from_` e `body` conterranno il numero del vostro cellulare, il numero di Twilio e il messaggio, rispettivamente. Notate che il numero di telefono mittente è nell'attributo `from_` (con un underscore alla fine), non `from`. Questo perché `from` in Python è una parola chiave (l'abbiamo vista utilizzata in una forma dell'enunciato `import`, per esempio, quella del tipo `from nomemodulo import *`), perciò non può essere utilizzata come nome di un attributo. Continuate l'esempio nella shell interattiva come segue:

```
>>> message.status  
'queued'  
>>> message.date_created  
datetime.datetime(2015, 7, 8, 1, 36, 18)  
>>> message.date_sent == None  
True
```

L'attributo `status` dovrebbe darvi una stringa. Gli attributi `date_created` e `date_sent` vi daranno un oggetto `datetime` se il messaggio è stato creato e inviato. Può sembrare strano che l'attributo `status` sia impostato a `'queued'` (accodato) e l'attributo `date_sent` sia impostato a `None` quando avete già ricevuto il messaggio, ma è così perché l'oggetto `Message` è stato catturato nella variabile `message` prima che il testo venisse effettivamente inviato. Dovrete recuperare di nuovo l'oggetto `Message` per vedere l'aggiornamento più recente di `status` e `date_sent`. Ogni messaggio di Twilio ha un ID univoco (SID) che può essere utilizzato per recuperare l'ultimo aggiornamento dell'oggetto `Message`. Continuate l'esempio nella shell interattiva inserendo quanto segue.

```
>>> message.sid  
'SM09520de7639ba3af137c6fcb7c5f4b51'  
❶ >>> updatedMessage = twilioCli.messages.get(message.sid)  
>>> updatedMessage.status  
'delivered'  
>>> updatedMessage.date_sent  
datetime.datetime(2015, 7, 8, 1, 36, 18)
```

Inserendo `message.sid` vi viene presentato questo lungo SID di messaggio. Passandolo al metodo `get()` del client Twilio ❶, potete recuperare un nuovo oggetto `Message` con le informazioni più aggiornate. In questo nuovo oggetto `Message`, gli attributi `status` e `date_sent` sono corretti.

L'attributo `status` sarà impostato a uno dei valori stringa seguenti: `'queued'`, `'sending'`, `'sent'`, `'delivered'`, `'undelivered'` o `'failed'`. Il significato dovrebbe essere chiaro, ma se volete i particolari più precisi potete consultare le risorse sul sito <http://nostarch.com/automatestuff/>.

Purtroppo, ricevere messaggi con Twilio è un po' più complicato che inviarli. Twilio richiede che abbiate un sito web che esegue la propria applicazione. Questo va al di là delle finalità del libro, ma potete trovare maggiori informazioni nelle risorse del libro all'indirizzo <http://nostarch.com/automatestuff/>.

## Progetto: un modulo “mandami solo un sms”

La persona a cui è più probabile mandate messaggi dai vostri programmi siete voi stessi. Gli sms sono un ottimo modo per spedire a voi stessi delle notifiche quando siete lontani dal vostro computer. Se avete automatizzato un compito particolarmente noioso con un programma, la cui esecuzione richiede un paio d'ore, potreste fare in modo che il programma vi mandi un sms quando ha finito il suo lavoro. Oppure potreste avere un programma che viene eseguito regolarmente in momenti predefiniti e che ha bisogno di contattarvi, magari un programma che consulta le previsioni del tempo e nel caso vi manda un sms per ricordarvi di prendere l'ombrelllo.

Come semplice esempio ecco un piccolo programma Python con una funzione `textmyself()` che invia un messaggio che viene passato come argomento stringa. Aprite una nuova finestra di file editor e inserite il codice seguente, sostituendo i segnaposto per SID dell'account, auth token e numeri telefonici con le vostre informazioni. Salvate il tutto con il nome `textMyself.py`.

```
#! python3
# textMyself.py - Definisce la funzione textmyself() che invia un sms
# con il testo che le viene passato come stringa.

# Valori predefiniti:
accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
authToken = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
myNumber = '+15559998888'
twilioNumber = '+15552225678'

from twilio.rest import TwilioRestClient

❶ def textmyself(message):
❷     twilioCli = TwilioRestClient(accountSID, authToken)
❸     twilioCli.messages.create(body=message, from_=twilioNumber, to=myNumber)
```

Questo programma memorizza SID di un account, auth token, numero del mittente e numero del destinatario, poi definisce `textmyself()` in modo che prenda un argomento ❶, crei un oggetto `TwilioRestClient` ❷ e chiami `create()` con il messaggio che le è stato passato ❸.

Se volete rendere la funzione `textmyself()` disponibile per altri programmi, semplicemente salvate il file `textMyself.py` nella stessa cartella dell'eseguibile Python. Ora potete usare la funzione nei vostri altri programmi. Ogni volta che volete che uno dei vostri programmi vi mandi un sms, basta che aggiungiate il codice seguente:

```
import textmyself
textmyself.textmyself('Il compito noioso è finito!')
```

Dovete iscrivervi a Twilio e scrivere il codice per gli sms solo una volta; poi bastano due righe di codice per inviare un sms da qualsiasi altro programma.

## Riepilogo

Comunichiamo via **Internet** e via **reti cellulari** in decine di modi diversi, ma la **posta elettronica** e gli **sms** sono le forme più comuni. I vostri programmi possono comunicare attraverso questi canali, e questo permette di dotarli di potenti caratteristiche di **notifica**. Potete addirittura scrivere programmi che girano su computer diversi e che comunicano fra loro direttamente via email, con un programma che invia messaggi con **SMTP** e l'altro che li riceve con **IMAP**.

Il modulo `smtplib` di Python mette a disposizione funzioni per usare SMTP per inviare email attraverso il server SMTP del vostro provider. Analogamente, i moduli di terze parti `imapclient` e `pyzmail` consentono di accedere a server IMAP e di recuperare email indirizzate a voi. IMAP è un po' più complicato di SMTP, ma è anche molto potente e consente di effettuare ricerche per recuperare determinati messaggi, scaricarli e analizzarli per estrarne l'oggetto e il corpo come valori stringa.

Gli sms sono un po' diversi dalla posta elettronica poiché, a differenza di quest'ultima l'invio di sms richiede qualcosa di più di una connessione Internet. Per fortuna, servizi come Twilio mettono a disposizione moduli che consentono di inviare sms dai programmi. Superato il processo iniziale di creazione di un account, potrete inviare messaggi semplicemente con un paio di righe di codice.

Con questi moduli nella vostra cassetta degli attrezzi, potrete programmare le condizioni specifiche sotto le quali i vostri programmi devono inviare notifiche o promemoria. Ora i vostri programmi possono estendersi molto oltre il computer su cui girano!

## Domande di ripasso

1. Qual è il protocollo per l'invio di email? E quello per controllare e ricevere email?
2. Quali funzioni/metodi del modulo `smtplib` dovete chiamare per accedere a un server SMTP?  
(Sono quattro.)
3. Quali due funzioni/metodi di `imapclient` dovete chiamare per accedere a un server IMAP?
4. Che tipo di argomento si passa a `imapObj.search()`?
5. Che cosa fate se il vostro codice riceve un messaggio di errore che dice got more than 1000 bytes?
6. Il modulo `imapclient` gestisce la connessione a un server IMAP e la ricerca di email. Qual è il modulo che gestisce la lettura delle email identificate da `imapclient`?
7. Quali sono le tre informazioni che dovete ottenere da Twilio per poter inviare sms?

## Un po' di pratica

Per esercitarvi, scrivete dei programmi che svolgano le attività seguenti.

## Email per l'assegnazione casuale di incombenze

Scrivete un programma che prenda una lista di indirizzi email di persone e una lista di incombenze da assolvere e abbini in modo casuale incombenze e persone, poi invii per email a ogni persona l'incombenza che le è stata assegnata. Se vi sentite ambiziosi, tenete traccia delle incombenze assegnate in precedenza a ciascuno, per far sì che il programma eviti di assegnare la stessa incombenza due volte di seguito alla stessa persona.

Un'altra possibile caratteristica è pianificare l'esecuzione automatica del programma una volta alla settimana.

Un suggerimento: se passate una lista alla funzione `random.choice()`, restituirà un elemento della lista scelto a caso. Una parte del codice potrebbe essere fatta in questo modo:

```
incombenze = ['lavare piatti', 'pulire bagno', 'passare aspirapolvere', 'portare il cane']
randomIncombenze = random.choice(incombenze)
incombenze.remove(randomIncombenze) # questa incombenza è già assegnata, togila dalla lista
```

## Non dimenticare l'ombrelllo

Nel [Capitolo 11](#) abbiamo visto come usare il modulo `requests` per estrarre dati da <http://weather.gov/>. Scrivete un programma che si avvii subito prima che vi alziate al mattino e controlli le previsioni del tempo. Se è prevista pioggia, il programma dovrà mandarvi un sms per ricordarvi di prendere con voi un ombrello.

## Cancellazione automatica delle iscrizioni

Scrivete un programma che esamini il vostro account di posta elettronica, trovi tutti i collegamenti per l'annullamento di iscrizioni nelle email e li apra automaticamente in un browser. Questo programma dovrà accedere al server IMAP del vostro provider e scaricare tutte le vostre email. Potete usare BeautifulSoup ([Capitolo 11](#)) per cercare ogni occorrenza della parola *unsubscribe* in un tag di collegamento ipertestuale HTML.

Quando avete una lista di questi URL, potete usare `webbrowser.open()` per aprire automaticamente tutti questi link in un browser.

Dovrete comunque esaminare manualmente e completare gli eventuali passi ulteriori per annullare le iscrizioni: nella maggior parte dei casi, questo comporta fare clic su un link per confermare.

Questo script però vi risparmia la fatica di dover esaminare tutte le vostre email alla ricerca dei link per cancellare l'iscrizione. Poi potete passare questo script ai vostri amici perché possano eseguirlo sui loro account di posta. (Fate solo attenzione a non memorizzare la password della vostra posta elettronica nel codice sorgente!)

## Controllare il computer attraverso l'email

Scrivete un programma che controlli un account di posta ogni 15 minuti per verificare se avete inviato istruzioni ed eventualmente le esegua automaticamente. Per esempio, BitTorrent è un sistema di download peer-to-peer. Utilizzando software BitTorrent gratuito come qBittorrent, potete scaricare sul vostro computer file mediatici di grandi dimensioni. Se inviate al vostro programma un link BitTorrent (del tutto legale, non pirata), il programma verificherà l'email, troverà il messaggio, estrarrà il link e lancerà qBittorrent per iniziare a scaricare il file. In questo modo, potete fare sì che il vostro computer inizi i download mentre non siete davanti alla tastiera e il download potrà essere completato quando tornerete a casa.

Nel [Capitolo 15](#) abbiamo visto come lanciare programmi utilizzando la funzione `subprocess.Popen()`. Per esempio, la chiamata seguente lancia il programma qBittorrent con un file torrent:

```
qbProcess = subprocess.Popen(['C:\\Program Files (x86)\\qBittorrent\\
qbittorrent.exe', 'shakespeare_complete_works.torrent'])
```

Ovviamente, vorrete che il programma verifichi che i messaggi provengono proprio da voi. In particolare, potreste richiedere che le email contengano una password, perché è abbastanza banale per un hacker simulare un indirizzo mittente nelle email. Il programma deve cancellare le email che identifica, in modo da non ripetere le istruzioni ogni volta che controlla l'account di posta. Come

caratteristica ulteriore, potete fare in modo che il programma vi mandi un messaggio di posta o un sms di conferma ogni volta che esegue un comando. Dato che non sarete davanti al computer che esegue il programma, è una buona idea usare le funzioni di logging ([Capitolo 10](#)) per scrivere un file di testo di log che poi potrete esaminare, nel caso si verifichi qualche errore.

qBittorrent (come altre applicazioni BitTorrent) ha una funzione che ne permette la chiusura automatica al completamento di un download. Il [Capitolo 15](#) spiega come si può stabilire quando un'applicazione lanciata sia stata chiusa, mediante il metodo `wait()` per oggetti `Popen`. Il metodo `wait()` sarà in un blocco fino a che qBittorrent non si sarà chiuso, poi il vostro programma può mandarvi una email o un sms per notificarvi il completamento del download.

Vi sono moltissime altre caratteristiche che potete aggiungere a un progetto del genere. Se vi trovate in difficoltà, potete scaricare una implementazione esemplificativa di questo programma da <http://nostarch.com/automatestuff/>.

# Manipolare immagini



Se avete una **fotocamera digitale** o semplicemente caricate foto su Facebook dal **cellulare**, probabilmente avrete a che fare costantemente con file di immagini digitali. Forse sapete usare qualche **programma di grafica** di base, come Microsoft Paint o Paintbrush, o magari anche applicazioni più raffinate come Adobe Photoshop. Se però dovete modificare un gran numero di immagini, intervenire manualmente può essere un compito lungo e noioso.

Qui entra in campo Python. Pillow è un modulo di terze parti per interagire con file di immagine. Questo modulo possiede varie funzioni grazie alle quali è facile ritagliare, ridimensionare e modificare i contenuti di un'immagine. Con la possibilità di manipolare immagini come fareste con software come Microsoft Paint o Adobe Photoshop, Python può modificare automaticamente centinaia o migliaia di immagini senza alcuna fatica.

## Nozioni fondamentali sulle immagini

Per poter manipolare un'immagine, dovete conoscere gli aspetti fondamentali del modo in cui i computer gestiscono colori e coordinate nelle immagini e del modo in cui si può lavorare con colori e coordinate in Pillow. Prima di continuare, però, installate il modulo pillow. (Potete consultare [l'Appendice A](#) per maggiori informazioni sull'installazione di moduli di terze parti.)

## Colori e valori RGBA

I programmi rappresentano spesso un **colore** in un'immagine come un **valore RGBA**. Un valore RGBA è un gruppo di numeri che specificano la quantità di rosso, verde, blu e alfa (o trasparenza) in un colore. Ciascuno di questi componenti è un intero compreso fra 0 (nulla) e 255 (il massimo). Questi valori RGBA sono assegnati ai singoli pixel, cioè i punti più piccoli di un unico colore che lo schermo di un computer può visualizzare (come potete immaginare, ci sono milioni di pixel su uno schermo). Le impostazioni RGB di un pixel dicono con precisione quale **sfumatura di colore** debba visualizzare. Le immagini hanno anche un **valore alfa**: se un'immagine è visualizzata sopra un'immagine di sfondo o sull'immagine del desktop, il valore alfa stabilisce in che misura lo sfondo rimane visibile al di sotto dei pixel dell'immagine.

In Pillow, i valori RGBA sono rappresentati da una tupla di quattro valori interi. Per esempio, il rosso è rappresentato da (255, 0, 0, 255). Questo colore ha il massimo di rosso, niente verde o blu e il massimo di alfa, il che significa che è completamente opaco. Il verde è rappresentato da (0, 255, 0, 255), il blu da (0, 0, 255, 255). Il bianco, combinazione di tutti i colori, è (255, 255, 255, 255), mentre il nero, che non ha alcun colore, è (0, 0, 0, 255).

Se un colore ha un valore alfa 0, è invisibile e in realtà non ha alcuna importanza quali siano i valori

RGB. In fin dei conti, un rosso invisibile ha lo stesso aspetto di un nero invisibile. Pillow usa i **nomi di colori standard** utilizzati anche dall'HTML. La Tabella 17.1 elenca una selezione di nomi di colori standard con i relativi valori.

**Tabella 17.1** - Nomi di colori standard e relativi valori RGBA.

Nome	Valore RGBA	Nome	Valore RGBA
White	(255, 255, 255, 255)	Red	(255, 0, 0, 255)
Green	(0, 128, 0, 255)	Blue	(0, 0, 255, 255)
Gray	(128, 128, 128, 255)	Yellow	(255, 255, 0, 255)
Black	(0, 0, 0, 255)	Purple	(128, 0, 128, 255)

Pillow offre la funzione `ImageColor.getcolor()` per non dover memorizzare i valori RGBA dei colori che volete usare. Questa funzione prende come primo argomento una stringa con il nome di un colore e come secondo argomento la stringa 'RGBA' e restituisce una tupla RGBA.

## Colori CMYK e RGB

A scuola avrete studiato che, mescolando rosso, giallo e blu, si possono ottenere gli altri colori: mescolando blu e giallo, per esempio, si ottiene il verde. Questo è il **modello sottrattivo** dei colori e vale per inchiostri, vernici, pigmenti. Questo è il motivo per cui le stampanti hanno cartucce di colore CMYK: ciano (blu), magenta (rosso), giallo e nero si possono mescolare in modo da ottenere qualsiasi colore.

La fisica della luce però usa quello che si chiama un **modello additivo** per i colori. Quando si combinano luci (come quelle emesse dallo schermo del computer) rosse, verdi e blu, si possono ottenere tutti gli altri colori. Per questo i valori RGB rappresentano i colori nei programmi per computer.

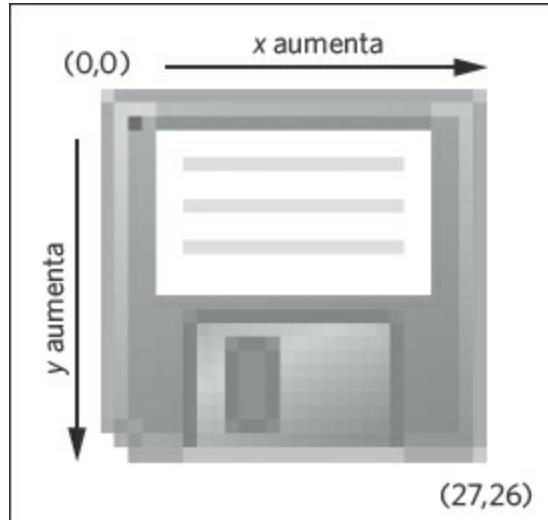
Per vedere come lavora questa funzione, inserite quanto segue nella shell interattiva:

```
❶ >>> from PIL import ImageColor
❷ >>> ImageColor.getcolor('red', 'RGBA')
(255, 0, 0, 255)
❸ >>> ImageColor.getcolor('RED', 'RGBA')
(255, 0, 0, 255)
>>> ImageColor.getcolor('Black', 'RGBA')
(0, 0, 0, 255)
>>> ImageColor.getcolor('chocolate', 'RGBA')
(210, 105, 30, 255)
>>> ImageColor.getcolor('CornflowerBlue', 'RGBA')
(100, 149, 237, 255)
```

Come prima cosa, dovete importare il modulo `ImageColor` da `PIL` ❶ (non da Pillow: vedremo il perché fra breve). La stringa con il nome del colore che si passa a `ImageColor.getcolor()` non fa distinzione fra maiuscole e minuscole, perciò passando 'red' ❷ e passando 'RED' ❸ si ottiene sempre la stessa tupla RGBA. Si possono passare anche nomi di colori meno comuni, come 'chocolate' o 'Cornflower Blue'. Pillow supporta un gran numero di nomi di colori, da 'aliceblue' a 'whitesmoke'. Potete trovare l'elenco completo degli oltre 100 nomi di colori standard nelle risorse all'indirizzo <http://nostarch.com/automatestuff/>.

## Coordinate e tuple box

I pixel delle immagini vengono identificati con **coordinate cartesiane**  $x$  e  $y$ , che specificano rispettivamente la posizione orizzontale e quella verticale di un pixel in un'immagine. L'origine è il pixel nell'angolo superiore destro dell'immagine e ha coordinate  $(0, 0)$ . Il primo zero rappresenta la coordinata  $x$ , che parte da zero all'origine e aumenta procedendo da sinistra verso destra; il secondo zero rappresenta la coordinata  $y$ , che inizia da zero all'origine e aumenta procedendo dall'alto verso il basso. Val la pena ripeterlo: le coordinate  $y$  aumentano procedendo verso il basso, che è il contrario di quel che magari ricordate dalle lezioni di matematica. La [Figura 17.1](#) dimostra come funziona questo sistema di coordinate.



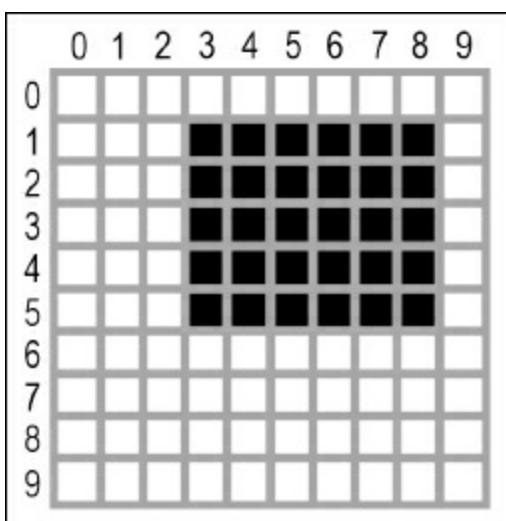
**Figura 17.1** - Le coordinate  $x$  e  $y$  di un'immagine di  $27 \times 26$  pixel che rappresenta un'antica forma di dispositivo di memoria.

Molte funzioni e molti metodi di Pillow prendono un argomento tupla box: questo significa che Pillow si aspetta una tupla con quattro coordinate intere che rappresentano una regione rettangolare in un'immagine.

I quattro interi sono, in ordine:

- *left*: la coordinata  $x$  del bordo a sinistra del riquadro;
- *top*: la coordinata  $y$  del bordo superiore del riquadro;
- *right*: la coordinata  $x$  di un pixel immediatamente a destra del bordo destro del riquadro. Questo intero deve essere maggiore dell'intero *left*;
- *bottom*: la coordinata  $y$  di un pixel immediatamente al di sotto del bordo inferiore del riquadro. Questo intero deve essere maggiore di *top*.

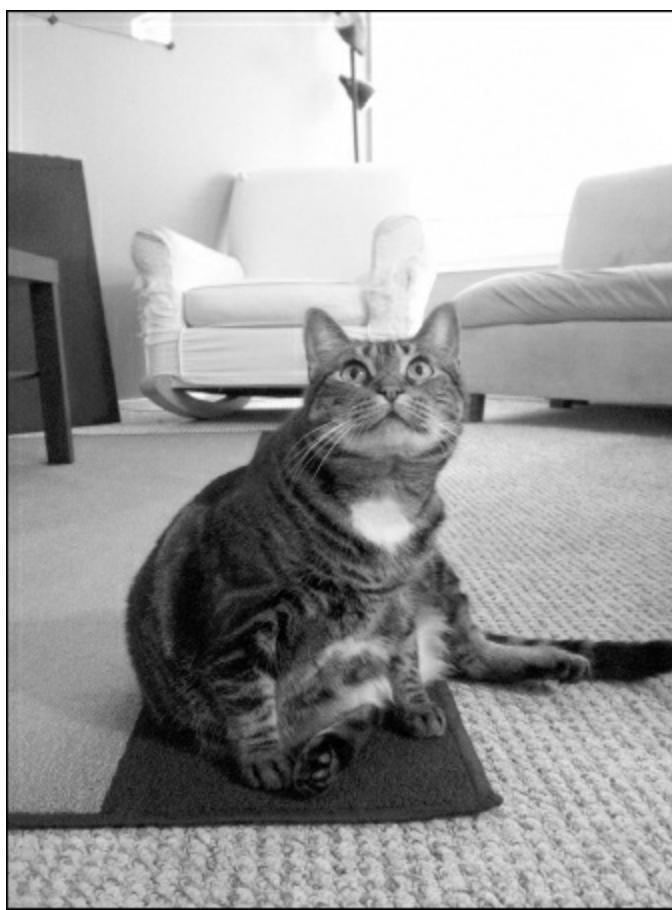
Notate che il box comprende le coordinate *left* e *top* ma non le coordinate *right* e *bottom*. Per esempio, la tupla  $(3, 1, 9, 6)$  rappresenta tutti i pixel nel riquadro nero della [Figura 17.2](#).



**Figura 17.2** - L'area in nero è rappresentata dalla tupla (3, 1, 9, 6).

## Manipolare immagini con Pillow

Ora che sapete come funzionano in Pillow colori e coordinate, usiamo questo modulo per manipolare un'immagine. La [Figura 17.3](#) rappresenta l'immagine che useremo per tutti gli esempi nella shell interattiva in questo capitolo. Potete scaricarla da <http://nostarch.com/automatestuff/>.



**Figura 17.3** - La mia gatta Zophie. (La fotocamera la ingrassa un po'...)

Quando avrete il file Zophie.png nella directory di lavoro corrente, sarete pronti per caricare l'immagine di Zophie in Python, in questo modo:

```
>>> from PIL import Image
```

```
>>> catIm = Image.open('zophie.png')
```

Per caricare l'immagine, importate il modulo Image da Pillow e chiamate `Image.open()` passandole il nome di file dell'immagine. Poi potete memorizzare l'immagine in una variabile come `CatIm`. Il nome del modulo Pillow è `PIL`, per ragioni di compatibilità con un modulo più vecchio che si chiamava **Python Imaging Library**: è questo il motivo per cui bisogna eseguire `from PIL import Image` anziché `from Pillow import Image`. Dato il modo in cui i suoi creatori hanno impostato il modulo `pillow`, dovete usare la forma `from Pillow import Image` dell'enunciato `import`, anziché un più semplice `import PIL`.

Se il file di immagine non si trova nella directory di lavoro corrente, cambiate quest'ultima in modo che coincida con la cartella che contiene il file di immagine, chiamando la funzione `os.chdir()`.

```
>>> import os  
>>> os.chdir('C:\\\\cartella_che_contiene_il_file')
```

La funzione `Image.open()` restituisce un valore del tipo di dati oggetto `Image`, ed è questo il modo in cui Pillow rappresenta un'immagine come valore Python. Potete caricare un oggetto `Image` da un file di immagine (di qualsiasi formato) passando alla funzione `Image.open()` una stringa con il nome del file. Qualsiasi cambiamento apportate all'oggetto `Image` può essere salvato in un file di immagine (di qualsiasi formato) con il metodo `save()`. Rotazioni, ridimensionamenti, ritagli, disegni e altri manipolazioni verranno effettuati mediante chiamate a metodi su questo oggetto `Image`.

Per abbreviare gli esempi in questo capitolo, assumo che abbiate importato il modulo `Image` di Pillow e che abbiate memorizzato l'immagine di Zophie in una variabile `catIm`. Controllate che il file `zophie.png` sia nella directory di lavoro corrente, in modo che la funzione `Image.open()` la possa trovare. Altrimenti, dovete anche specificare il percorso assoluto completo nell'argomento stringa di `Image.open()`.

## Lavorare con il tipo di dati immagine

Un oggetto `Image` possiede vari **attributi** utili che forniscono **informazioni** fondamentali sul file di immagine da cui è stato caricato: larghezza e altezza, nome di file e formato grafico (per esempio JPEG, GIF o PNG).

Per esempio, inserite quanto segue nella shell interattiva:

```
>>> from PIL import Image  
>>> catIm = Image.open('zophie.png')  
>>> catIm.size  
❶ (816, 1088)  
❷ >>> width, height = catIm.size  
❸ >>> width  
816  
❹ >>> height  
1088  
>>> catIm.filename  
'zophie.png'  
>>> catIm.format  
'PNG'  
>>> catIm.format_description  
'Portable network graphics'  
❺ >>> catIm.save('zophie.jpg')
```

Dopo aver creato un oggetto `Image` da `Zophie.png` e aver memorizzato l'oggetto `Image` in `catIm`, possiamo vedere che l'attributo `size` dell'oggetto contiene una tupla con la larghezza e l'altezza in pixel dell'immagine ❶. Possiamo assegnare i valori nella tupla a variabili `width` e `height` ❷, per poter accedere individualmente a larghezza ❸ e altezza ❹. L'attributo `filename` descrive il nome originale del file. Gli attributi `format` e `format_description` sono stringhe che descrivono il formato del file originale (il secondo è un po' più prolisso).

Infine, chiamando il metodo `save()` e passandogli '`zophie.jpg`' si salva sul disco fisso una nuova immagine con il nome di file `zophie.jpg` ❺. Pillow vede che l'estensione è `.jpg` e salva automaticamente l'immagine nel formato JPEG. Ora sul disco fisso avrete due immagini, `zophie.png` e `zophie.jpg`: i due file si basano sulla stessa immagine, ma non sono identici perché sono di formati diversi.

Pillow mette a disposizione anche la funzione `Image.new()`, che restituisce un oggetto `Image`, in modo simile a `Image.open()`, tranne che l'immagine rappresentata dall'oggetto `Image.new()` sarà vuota. Gli argomenti di `Image.new()` sono i seguenti:

- la stringa '`RGBA`', che imposta la modalità colore a `RGBA` (esistono anche altre modalità, ma qui non ce ne occuperemo);
- le dimensioni, sotto forma di tupla di due interi con altezza e larghezza della nuova immagine;
- il colore di sfondo con cui deve aprirsi l'immagine, sotto forma di tupla con i quattro interi che danno un valore `RGB`. Potete usare il valore restituito dalla funzione `ImageColor.getcolor()` per questo argomento; in alternativa, `Image.new()` ammette anche il passaggio della stringa con il nome di un colore standard.

Per esempio, inserite quanto segue nella shell interattiva:

```
❶ >>> from PIL import Image  
❶ >>> im = Image.new('RGBA', (100, 200), 'purple')  
❶ >>> im.save('purpleImage.png')  
❷ >>> im2 = Image.new('RGBA', (20, 20))  
❷ >>> im2.save('transparentImage.png')
```

Qui creiamo un oggetto `Image` per un'immagine larga 100 pixel e alta 200 con uno sfondo viola (`purple`) ❶. Questa immagine poi viene salvata nel file `purpleImage.png`. Chiamiamo poi di nuovo `Image.new()` per creare un altro oggetto `Image`, questa volta passando `(20, 20)` per le dimensioni e nulla per il colore di sfondo ❷. Il nero invisibile, `(0, 0, 0, 0)` è il colore predefinito utilizzato quando non viene specificato alcun argomento colore, perciò la seconda immagine ha uno sfondo trasparente; salviamo il quadrato trasparente  $20 \times 20$  in `transparentImage.png`.

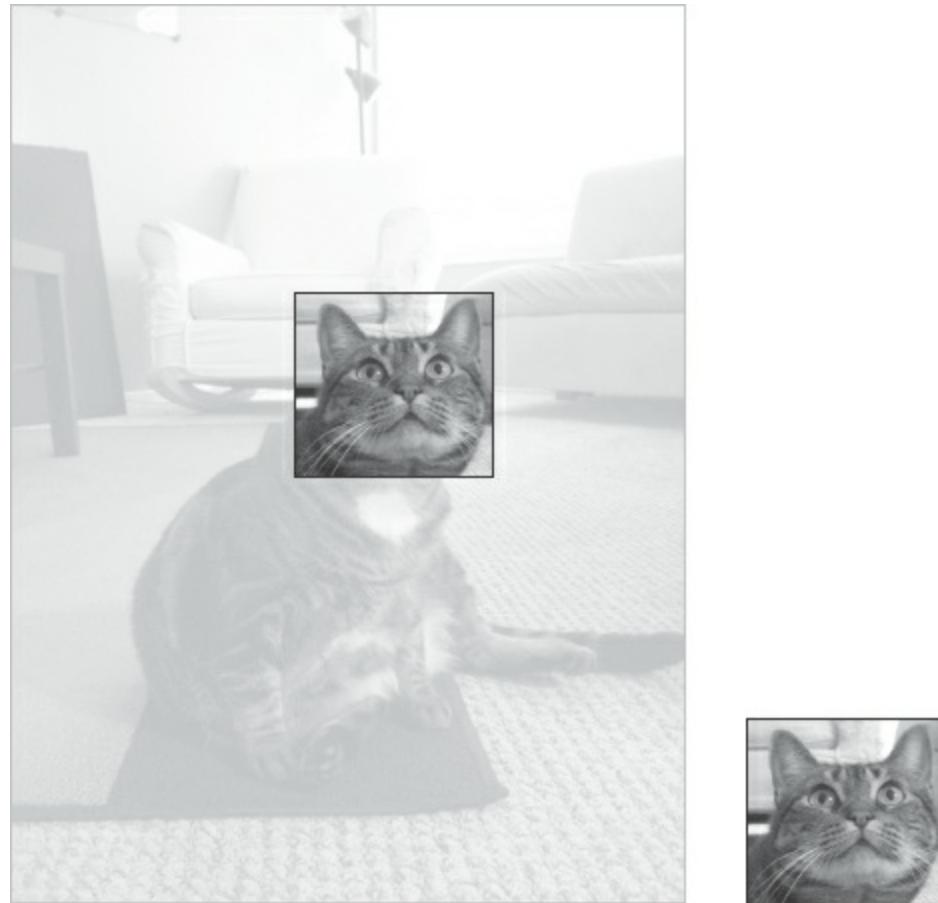
## Ritagliare immagini

**Ritagliare (crop)** un'immagine significa selezionare una regione rettangolare all'interno dell'immagine ed eliminare tutto ciò che cade al di fuori di quel rettangolo. Il metodo `crop()` su oggetti `Image` prende una tupla di box e restituisce un oggetto `Image` che rappresenta l'immagine ritagliata. Il ritaglio non avviene "sul posto": l'oggetto `Image` originale, cioè, non viene toccato e il metodo `crop()` restituisce un nuovo oggetto `Image`. Ricordate che una tupla di box (in questo caso, la sezione ritagliata) comprende la colonna di sinistra e la fila superiore di pixel, ma non include la colonna di destra e la riga inferiore di pixel.

Inserite quanto segue nella shell interattiva:

```
>>> croppedIm = catIm.crop((335, 345, 565, 560))  
>>> croppedIm.save('cropped.png')
```

Questo crea un nuovo oggetto `Image` per l'immagine ritagliata, memorizza l'oggetto in `croppedIm`, poi chiama `save()` su `croppedIm` per salvare l'immagine ritagliata in `cropped.png`. Il nuovo file viene creato a partire dall'immagine originale, come nella [figura 17.4](#).



**Figura 17.4** - La nuova immagine sarà la sezione ritagliata dell'immagine originale.

## Copiare e incollare immagini su altre immagini

Il metodo `copy()` restituirà un nuovo oggetto `Image` con la stessa immagine dell'oggetto `Image` su cui è stato chiamato. Questo metodo è utile se dovete modificare un'immagine ma volete conservare anche una versione immutata dell'originale. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> catIm = Image.open('zophie.png')  
>>> catCopyIm = catIm.copy()
```

Le variabili `catIm` e `catCopyIm` contengono due distinti oggetti `Image`, che contengono la stessa immagine. Ora che avete un oggetto `Image` memorizzato in `catCopyIm`, potete modificare `catCopyIm` come volete e salvare il risultato con un nuovo nome di file, lasciando immutato `zophie.png`. Per esempio, proviamo a modificare `catCopyIm` con il metodo `paste()`.

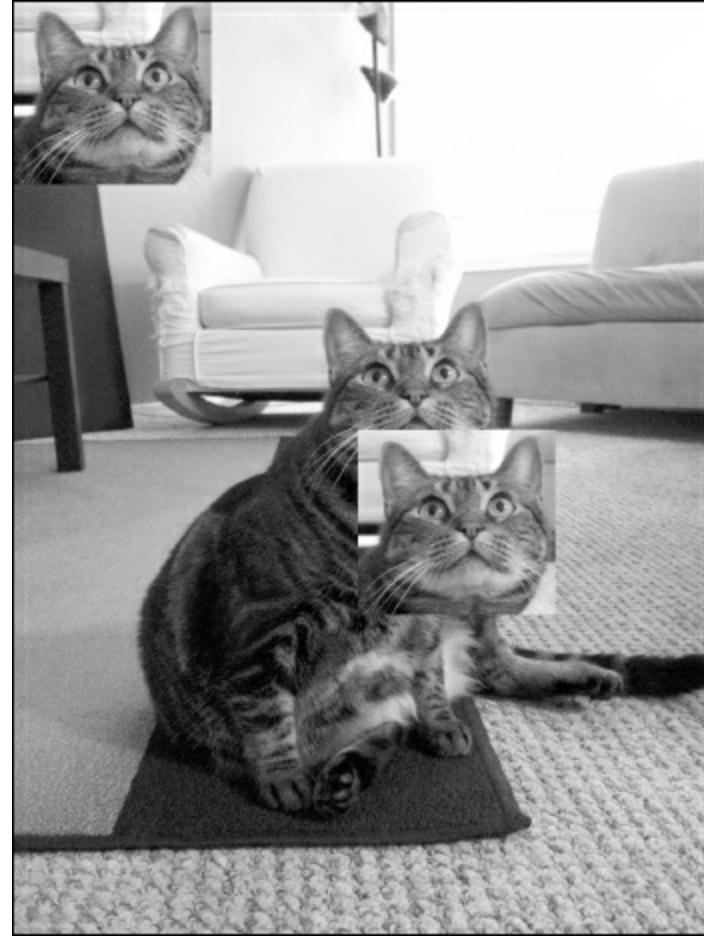
Il metodo `paste()` viene chiamato su un oggetto `Image` e incolla un'altra immagine su questa. Continuiamo l'esempio della shell incollando un'immagine più piccola su `catCopyIm`.

```
>>> faceIm = catIm.crop((335, 345, 565, 560))  
>>> faceIm.size
```

(230, 215)

```
>>> catCopyIm.paste(faceIm, (0, 0))
>>> catCopyIm.paste(faceIm, (400, 500))
>>> catCopyIm.save('pasted.png')
```

Prima passiamo a `crop()` una tupla per l'area rettangolare di `zophie.png` che contiene il muso di Zophie. Questo crea un oggetto `Image` che rappresenta un ritaglio di  $230 \times 215$  pixel, che memorizziamo in `faceIm`. Ora possiamo incollare `faceIm` su `catCopyIm`. Il metodo `paste()` richiede due argomenti: un oggetto `Image` "sorgente" e una tupla con le coordinate  $x$  e  $y$  del punto in cui si vuole collocare l'angolo superiore sinistro dell'oggetto `Image` sorgente sull'oggetto `Image` principale. Qui chiamiamo `paste()` due volte su `catCopyIm`, passando la prima volta  $(0, 0)$  e la seconda  $(400, 500)$ . Questo incolla `faceIm` su `catCopyIm` due volte: una volta con l'angolo superiore sinistro di `faceIm` in  $(0, 0)$  su `catCopyIm`, e una volta con l'angolo superiore sinistro di `faceIm` in  $(400, 500)$ . Infine, salviato `catCopyIm` modificato in `pasted.png`. Questa immagine sarà come quella visibile nella [Figura 17.5](#).



**Figura 17.5** - Zophie, con il muso incollato due volte.

## NOTA

Nonostante i nomi i metodi `copy()` e `paste()` in Pillow non fanno uso degli Appunti del computer.

Notate che il metodo `paste()` modifica il suo oggetto `Image` sul posto: non restituisce un oggetto `Image` con l'immagine incollata. Se volete chiamare `paste()` ma conservare anche una versione inalterata dell'immagine originale, dovete prima copiare l'immagine e poi chiamare `paste()` sulla copia.

Supponiamo che vogliate tassellare tutta l'immagine con il muso di Zophie, come nella [Figura 17.6](#). Potete ottenere questo effetto semplicemente con un paio di cicli for. Continuate l'esempio nella shell interattiva inserendo quanto segue:

```
>>> catImWidth, catImHeight = catIm.size
>>> faceImWidth, faceImHeight = faceIm.size
❶ >>> catCopyTwo = catIm.copy()
❷ >>> for left in range(0, catImWidth, faceImWidth):
❸     for top in range (0, catImHeight, faceImHeight):
         print(left, top)
         catCopyTwo.paste(faceIm, (left, top))

0 0
0 215
0 430
0 645
0 860
0 1075
230 0
230 215
--righe omesse--
690 860
690 1075
>>> catCopyTwo.save('tiled.png')
```



**Figura 17.6** - Cicli for annidati utilizzati con paste() duplicano il muso del gatto (si ottiene così un dupligatto, se volete).

Qui memorizziamo larghezza e altezza di catIm in catImWidth e catImHeight. In ❶ facciamo una copia di catIm e la memorizziamo in catCopyTwo. Ora che abbiamo una copia su cui possiamo incollare,

cominciamo un ciclo per incollare `faceIm` su `catCopyTwo`. La variabile `left` del ciclo for più esterno inizia da 0 e procede per incrementi di `faceImWidth(230)` ❷. La variabile `top` del ciclo for interno inizia da 0 e procede per incrementi di `faceImHeight(225)` ❸. Questi cicli for annidati producono valori di `left` e `top` per incollare una griglia di immagini `faceIm` sull'oggetto `Image` `catCopyTwo`, come nella [Figura 17.6](#). Per vedere come operano i cicli annidati, stampiamo `left` e `top`. Completate le operazioni di incollaggio, salviamo `catCopyTwo` modificata in `tiled.png`.

## Incollare pixel trasparenti

Normalmente i pixel trasparenti vengono incollati come pixel bianchi. Se l'immagine che volete incollare ha pixel trasparenti, passate l'oggetto `Image` come terzo argomento in modo che non venga incollato un rettangolo opaco. Questo terzo argomento è l'oggetto `Image` "maschera". Una maschera è un oggetto `Image` in cui il valore alfa è significativo, mentre sono ignorati i valori rosso, verde e blu. La maschera dice alla funzione `paste()` quali pixel deve copiare e quali lasciare trasparenti. Un uso avanzato delle maschere va al di là dei limiti di questo libro, ma se volete incollare un'immagine che ha pixel trasparenti, passate di nuovo l'oggetto `Image` come terzo argomento.

## Ridimensionare un'immagine

Il metodo `resize()` si chiama su un oggetto `Image` e restituisce un nuovo oggetto `Image` della larghezza e altezza specificate. Accetta come argomento una tupla di due interi, che rappresentano larghezza e altezza dell'immagine restituita. Inserite quanto segue nella shell interattiva:

```
❶ >>> width, height = catIm.size  
❷ >>> quartersizedIm = catIm.resize((int(width / 2), int(height / 2)))  
     >>> quartersizedIm.save('quartersized.png')  
❸ >>> svelteIm = catIm.resize((width, height + 300))  
     >>> svelteIm.save('svelte.png')
```

Qui assegniamo i due valori nella tupla `catIm.size` alle variabili `width` e `height` ❶. Utilizzando `width` e `height` invece che `catIm.size[0]` e `catIm.size[1]`, il resto del codice diventa più leggibile.

La prima chiamata `resize()` passa `int(width / 2)` come nuova larghezza e `int(height / 2)` come nuova altezza ❷, in modo che l'oggetto `Image` restituito da `resize()` abbia larghezza e altezza metà dell'immagine originale (la superficie sarà quindi un quarto). Il metodo `resize()` accetta nel suo argomento tupla solo interi, ed è questo il motivo per cui è stato necessario racchiudere entrambe le divisioni per 2 in una chiamata `int()`.

Questo ridimensionamento mantiene le stesse proporzioni per larghezza e altezza, ma le nuove dimensioni passate a `resize()` non devono per forza essere proporzionali all'immagine originale. La variabile `svelteIm` contiene un oggetto `Image` che ha la larghezza originale, ma un'altezza maggiore di 300 pixel ❸, il che rende Zophie molto più slanciata.

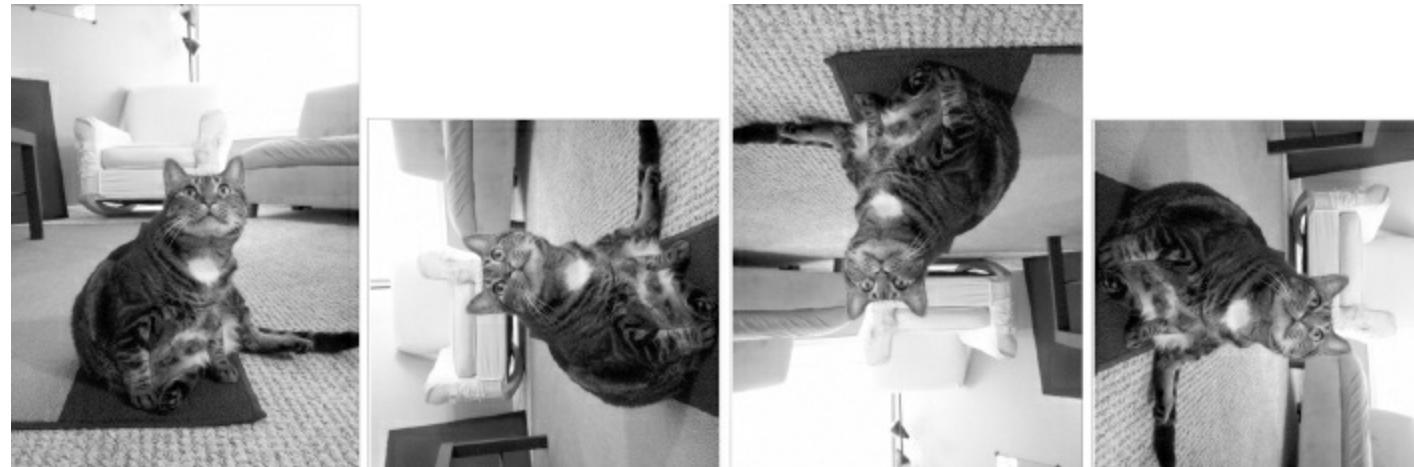
## Ruotare e ribaltare immagini

Si possono ruotare le immagini con il metodo `rotate()`, che restituisce un nuovo oggetto `Image` dell'immagine ruotata e lascia intatto l'oggetto `Image` originale. L'argomento di `rotate()` è un singolo valore intero o in virgola mobile che rappresenta il numero di gradi di cui si deve ruotare l'immagine, in senso antiorario. Inserite quanto segue nella shell interattiva:

```
>>> catIm.rotate(90).save('rotated90.png')
```

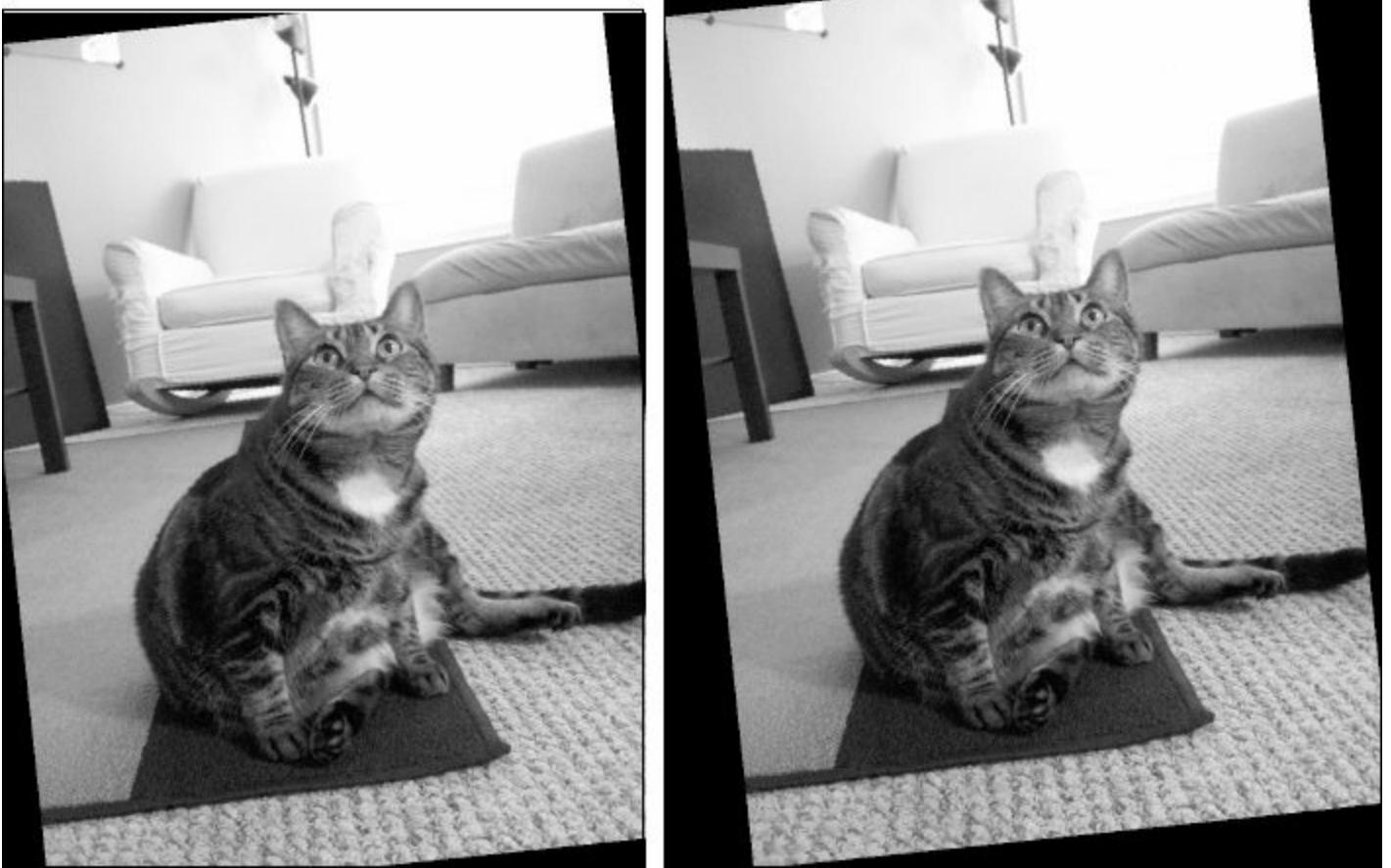
```
>>> catIm.rotate(180).save('rotated180.png')
>>> catIm.rotate(270).save('rotated270.png')
```

Notate come sia possibile concatenare le chiamate dei metodi, chiamando `save()` direttamente sull'oggetto `Image` restituito da `rotate()`. La prima chiamata a `rotate()` e `save()` crea un nuovo oggetto `Image` che rappresenta l'immagine ruotata di 90 gradi in senso antiorario e salva l'immagine ruotata in `rotated90.png`. Le due chiamate successive fanno la stessa cosa, ma ruotando di 180 e 270 gradi rispettivamente. I risultati sono come nella [Figura 17.7](#).



**Figura 17.7** - L'immagine originale (a sinistra) e l'immagine ruotata in senso antiorario di 90, 180 e 270 gradi, rispettivamente.

Notate che larghezza e altezza dell'immagine cambiano, quando l'immagine viene ruotata di 90 o 270 gradi. Se ruotate un'immagine di qualche altro angolo, le dimensioni originali vengono mantenute. In Windows, viene mantenuto uno sfondo nero per riempire le eventuali lacune lasciate dalla rotazione, come nella [Figura 17.8](#). In OS X, invece, vengono utilizzati pixel trasparenti.



**Figura 17.8** - L'immagine ruotata di 6 gradi normalmente (a sinistra) e con expand=True (a destra).

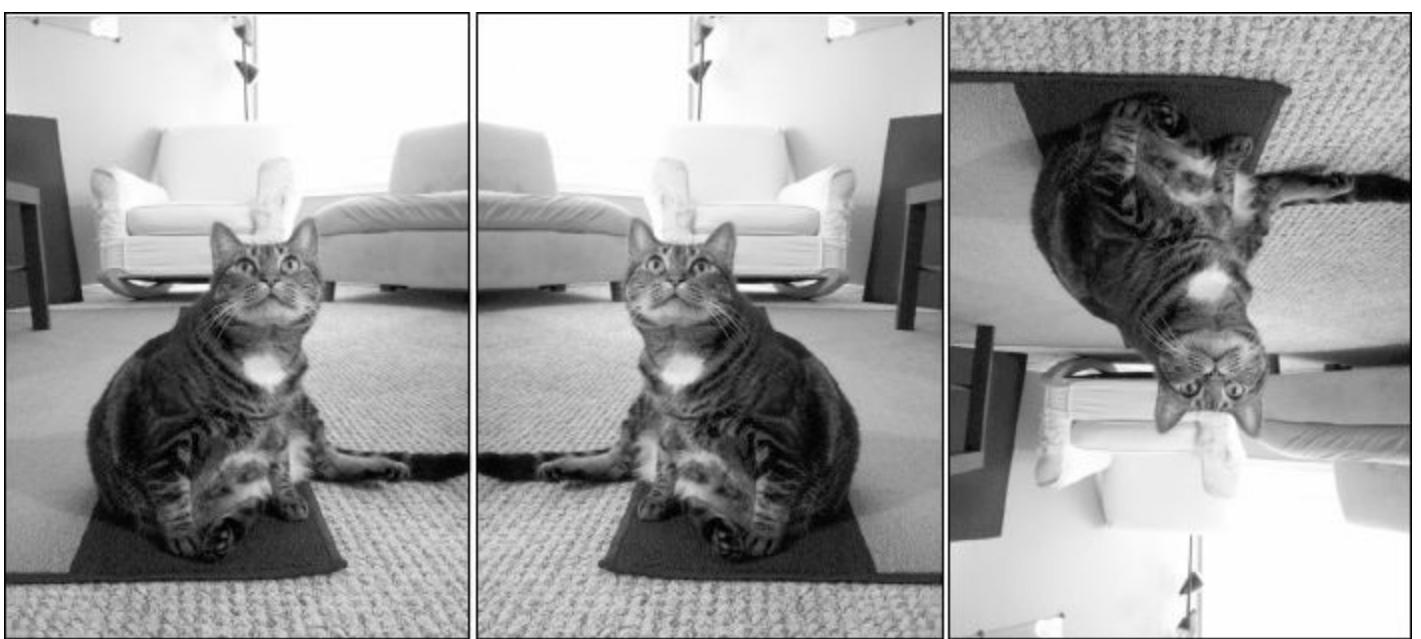
Il metodo `rotate()` ha un argomento facoltativo per parola chiave `expand`, che può essere impostato a `True` per ingrandire le dimensioni dell'immagine in modo che si adatti a tutta la nuova immagine ruotata. Per esempio, inserite quanto segue nella shell interattiva:

```
>>> catIm.rotate(6).save('rotatedd6.png')
>>> catIm.rotate(6, expand=True).save('rotatedd6_expanded.png')
```

Potete anche ribaltare specularmente un'immagine con il metodo `transpose()`. Dovete passare al metodo `transpose()` o `Image.FLIP_LEFT_RIGHT` o `Image.FLIP_TOP_BOTTOM`. Inserite quanto segue nella shell interattiva:

```
>>> catIm.transpose(Image.FLIP_LEFT_RIGHT).save('horizontal_flip.png')
>>> catIm.transpose(Image.FLIP_TOP_BOTTOM).save('vertical_flip.png')
```

Come `rotate()`, anche `transpose()` crea un nuovo oggetto `Image`. Qui passiamo `Image.FLIP_LEFT_RIGHT` per ribaltare l'immagine in orizzontale, poi salviamo il risultato in `horizontal_flip.png`; per ottenere l'immagine speculare in verticale, passiamo `Image.FLIP_TOP_BOTTOM` e salviamo il risultato in `vertical_flip.png`. I risultati sono visibili nella [Figura 17.9](#).



**Figura 17.9** - L'immagine originale (a sinistra), ribaltata in orizzontale (al centro) e in verticale (a destra).

## Modificare singoli pixel

Il colore di un singolo pixel può essere recuperato o impostato con i metodi `getpixel()` e `putpixel()`. Entrambi questi metodi prendono una tupla che rappresenta le coordinate  $x$  e  $y$  del pixel.

Il metodo `putpixel()` prende anche un ulteriore argomento tupla per il colore del pixel, che deve essere una tupla RGBA di quattro interi o una tupla RGB di tre interi.

Inserite quanto segue nella shell interattiva:

```

❶ >>> im = Image.new('RGBA', (100, 100))
❷ >>> im.getpixel((0, 0))
(0, 0, 0, 0)
❸ >>> for x in range(100):
    for y in range(50):
        im.putpixel((x, y), (210, 210, 210))

❹ >>> from PIL import ImageColor
❺ >>> for x in range(100):
    for y in range(50, 100):
        im.putpixel((x, y), ImageColor.getcolor('darkgray', 'RGBA'))
❻ >>> im.getpixel((0, 0))
(210, 210, 210, 255)
>>> im.getpixel((0, 50))
(169, 169, 169, 255)
>>> im.save('putPixel.png')

```

In ❶ creiamo una nuova immagine, un quadrato trasparente  $100 \times 100$ . Chiamando `getpixel()` su alcune coordinate in questa immagine si ha di ritorno  $(0, 0, 0, 0)$ , perché l'immagine è trasparente ❷. Per colorare i pixel in questa immagine, possiamo usare cicli `for` annidati per percorrere tutti i pixel nella metà superiore dell'immagine ❸ e colorare ciascun pixel mediante `putpixel()` ❹. Qui passiamo a `putpixel()` la tupla RGB  $(210, 210, 210)$ , che corrisponde a un grigio chiaro.

Supponiamo di volere come colore per la metà inferiore dell'immagine un grigio scuro, ma che non sappiamo quale sia la tupla RGB per questo colore. Il metodo `putpixel()` non accetta un nome di colore standard come `'darkgray'`, perciò bisogna usare `ImageColor.getcolor()` per ottenere una tupla di colore da

'darkgray'. Cicliamo sui pixel nella metà inferiore dell'immagine ❸ e passiamo a `putpixel()` il valore restituito da `ImageColor.getcolor()` ❹; si dovrebbe ottenere così un'immagine che è un rettangolo grigio chiaro in alto e uno grigio più scuro in basso, come nella [Figura 17.10](#). Potete chiamare `getpixel()` su qualche coordinata per verificare che il colore per ogni dato pixel sia proprio quello che vi aspettavate. Infine, salvate l'immagine in `putPixel.png`.



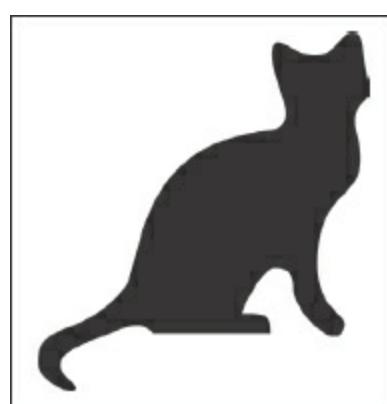
**Figura 17.10** - L'immagine `putPixel.png`.

Ovviamente, disegnare un pixel alla volta su un'immagine non è molto comodo. Se dovete disegnare delle forme, usate le funzioni `ImageDraw`, di cui parliamo più avanti in questo capitolo.

## Progetto: aggiungere un logo

Supponiamo che abbiate il noioso compito di ridimensionare migliaia di immagini, aggiungendovi un piccolo logo come watermark in un angolo. Una cosa simile con un programma di grafica semplice come Paintbrush o Paint richiederebbe un'infinità di tempo. Un'applicazione più sofisticata come Photoshop può fare elaborazioni batch, ma si tratta di software che costa centinaia di euro. Proviamo a scrivere invece uno script in grado di svolgere questo compito.

Supponiamo che quello nella [Figura 17.11](#) sia il logo che volete aggiungere nell'angolo inferiore destro di ciascuna immagine: l'icona di un gatto in nero con un bordo bianco e il resto dell'immagine trasparente.



**Figura 17.11** - Il logo da aggiungere alle immagini.

Ad alto livello, ecco cosa deve fare il programma:

- caricare l'immagine del logo;
- ciclare su tutti i file .png e .jpg nella directory di lavoro;
- controllare se l'immagine sia più larga o più alta di 300 pixel;
- se lo è, ridurre la larghezza o l'altezza (quale che sia la dimensione maggiore) a 300 pixel e

ridurre l'altra dimensione in proporzione;

- incollare il logo nell'angolo;
- salvare le immagini modificate in un'altra cartella.

Questo significa che il codice dovrà fare queste cose:

- aprire il file *catlogo.png* come oggetto Image;
- ciclare sulle stringhe restituite da os. listdir('');
- ottenere larghezza e altezza dell'immagine dall'attributo size;
- calcolare la nuova larghezza e altezza dell'immagine ridimensionata;
- chiamare il metodo `resize()` per ridimensionare l'immagine;
- chiamare il metodo `paste()` per incollare il logo;
- chiamare il metodo `save` per salvare le modifiche, usando il nome di file originale.

## Passo 1: aprire l'immagine del logo

Per questo progetto, apriete una nuova finestra di file editor, inserite il codice seguente e salvatelo con il nome *resizeAndAddLogo.py*.

```
#! python3
# resizeAndAddLogo.py - Ridimensiona tutte le immagini nella directory
# di lavoro corrente, in modo che stiano in un quadrato 300x300,
# e aggiunge catlogo.png nell'angolo in basso a destra.

import os
from PIL import Image

❶ SQUARE_FIT_SIZE = 300
❷ LOGO_FILENAME = 'catlogo.png'

❸ logoIm = Image.open(LOGO_FILENAME)
❹ logoWidth, logoHeight = logoIm.size

# DA FARE: Ciclare su tutti i file nella directory di lavoro.

# DA FARE: Verificare se l'immagine deve essere ridimensionata.

# DA FARE: Calcolare la nuova larghezza e altezza a cui ridimensionarla.

# DA FARE: Ridimensionare l'immagine.

# DA FARE: Aggiungere il logo.

# DA FARE: Salvare le modifiche.
```

Abbiamo impostato le **costanti** `SQUARE_FIT_SIZE` e `LOGO_FILENAME` all'inizio del programma, in modo che sia facile modificarlo eventualmente in seguito. Supponiamo che il logo non sia l'icona del gatto, o che dobbiate ridurre la dimensione maggiore dell'immagine a qualche valore diverso da 300 pixel: con queste costanti a inizio programma, potete semplicemente aprire il codice, modificare una volta questi valori e basta. (Oppure potete fare in modo che i valori di queste costanti vengano

recuperati dagli argomenti della riga di comando.) Senza queste costanti, dovreste invece cercare nel codice tutti i punti in cui si trovano 300 e 'catlogo.png' e sostituirle con i valori giusti per il nuovo progetto. In sintesi: l'uso delle costanti rende il programma più generale.

L'oggetto `Image` del logo viene restituito da `Image.open()` ❸. Per ragioni di leggibilità, a `logoWidth` e `logoHeight` vengono assegnati i valori di `logoIm.size` ❹.

Il resto per ora è solo uno scheletro di commenti con le cose DA FARE.

## Passo 2: ciclare su tutti i file e aprire le immagini

Ora dovrete trovare tutti i file `.png` e `.jpg` nella directory di lavoro corrente. Notate che non va aggiunta l'immagine del logo all'immagine del logo stessa, perciò il programma deve escludere qualsiasi immagine con un nome di file identico a quello che si trova in `LOGO_FILENAME`. Aggiungete quanto segue al vostro codice:

```
#! python3
# resizeAndAddLogo.py - Ridimensiona tutte le immagini nella directory
# di lavoro corrente, in modo che stiano in un quadrato 300x300,
# e aggiunge catlogo.png nell'angolo in basso a destra.

import os
from PIL import Image

--righe omesse--

os.makedirs('withLogo', exist_ok=True)
# Cicla su tutti i file nella directory di lavoro corrente.
❶ for filename in os.listdir('.'):
❷     if not (filename.endswith('.png') or filename.endswith('.jpg')) \
        or filename == LOGO_FILENAME:
❸         continue # skip non-image files and the logo file itself

❹     im = Image.open(filename)
    width, height = im.size

--righe omesse--
```

Come prima cosa, la chiamata a `os.makedirs()` crea una cartella `withLogo` in cui salvare le immagini completate con il logo, anziché sovrascrivere i file originali. L'argomento per parola chiave `exist_ok=True` impedirà a `os.makedirs()` di sollevare un'eccezione se `withLogo` esiste già. Mentre si cicla sui file nella directory di lavoro con `os.listdir('.')` ❶, il lungo enunciato `if` ❷ controlla se il nome di file non finisce con `.png` o `.jpg`. In quel caso, o se il file è la stessa immagine del logo, il ciclo lo deve saltare e usa `continue` ❸ per andare al file successivo. Se `filename` finisce con `.png` o `.jpg` (e non è il file del logo), può aprirlo come oggetto `Image` ❹ e impostarne `width` e `height`.

## Passo 3: ridimensionare le immagini

Il programma deve ridimensionare l'immagine solo se la sua larghezza o la sua altezza sono maggiori di `SQUARE_FIT_SIZE` (300 pixel, in questo caso), perciò mettete tutto il codice per il ridimensionamento all'interno di un enunciato `if` che verifica le variabili `width` e `height`. Aggiungete il codice seguente al vostro programma:

```

#! python3
# resizeAndAddLogo.py - Ridimensiona tutte le immagini nella directory
# di lavoro corrente, in modo che stiano in un quadrato 300x300,
# e aggiunge catlogo.png nell'angolo in basso a destra.

import os
from PIL import Image

--righe omesse--

# Verifica se l'immagine deve essere ridimensionata.
if width > SQUARE_FIT_SIZE and height > SQUARE_FIT_SIZE:
    # Calcola i nuovi valori di larghezza e altezza a cui ridimensionarla.
    if width > height:
        ❶      height = int((SQUARE_FIT_SIZE / width) * height)
        width = SQUARE_FIT_SIZE
    else:
        ❷      width = int((SQUARE_FIT_SIZE / height) * width)
        height = SQUARE_FIT_SIZE

    # Ridimensiona l'immagine.
    print('Sto ridimensionando %s...' % (filename))
    ❸      im = im.resize((width, height))

--righe omesse--

```

Se l'immagine deve essere ridimensionata, dovete scoprire se è troppo larga o troppo alta. Se `width` è maggiore di `height`, allora l'altezza deve essere ridotta in proporzione alla riduzione della larghezza ❶. Questa proporzione è il valore di `SQUARE_FIT_SIZE` diviso per la larghezza corrente. Il nuovo valore di `height` è questa proporzione moltiplicata per il valore corrente di `height`.

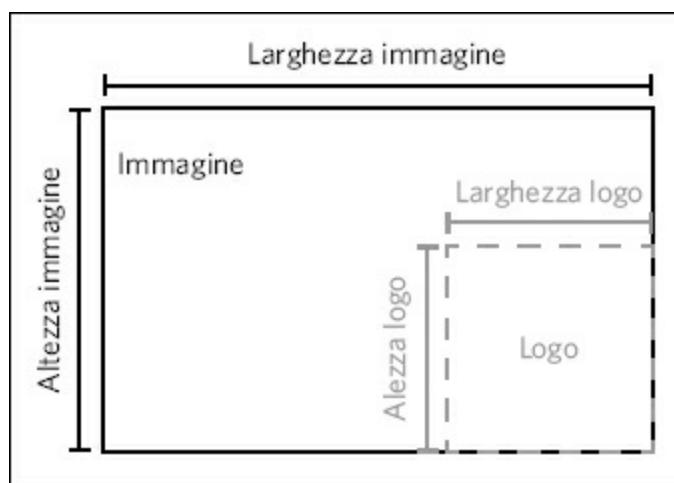
Dato che l'operatore di divisione restituisce un valore in virgola mobile e `resize()` richiede che le dimensioni siano interi, ricordare di convertire il risultato a un intero con la funzione `int()`. Infine, il nuovo valore di `width` verrà semplicemente impostato a `SQUARE_FIT_SIZE`.

Se `height` è maggiore o uguale a `width` (entrambi i casi sono gestiti nella clausola `else`), allora viene effettuato lo stesso calcolo, ma scambiando fra loro le variabili `height` e `width` ❷.

Quando `width` e `height` contengono le nuove dimensioni dell'immagine, le passate al metodo `resize()` e memorizzate l'oggetto `Image` restituito in `im` ❸.

## Passo 3: aggiungere il logo e salvare le modifiche

Indipendentemente dal fatto che l'immagine abbia dovuto essere ridimensionata o meno, bisogna ancora incollare il logo nell'angolo inferiore destro. Dove esattamente debba essere incollato dipende sia dalle dimensioni dell'immagine, sia dalle dimensioni del logo. La Figura 17.12 mostra come calcolare la posizione in cui incollare. La coordinata di sinistra per la posizione in cui incollare il logo sarà uguale alla larghezza dell'immagine meno la larghezza del logo; la coordinata superiore sarà l'altezza dell'immagine meno l'altezza del logo.



**Figura 17.12** - Le coordinate per collocare il logo nell'angolo inferiore destro devono essere la larghezza/altezza dell'immagine meno la larghezza/altezza del logo.

Dopo che il codice ha incollato il logo sull'immagine, bisogna salvare l'oggetto `Image` modificato. Aggiungete quanto segue al vostro programma:

```
#! python3
# resizeAndAddLogo.py - Ridimensiona tutte le immagini nella directory
# di lavoro corrente, in modo che stiano in un quadrato 300x300,
# e aggiunge catlogo.png nell'angolo in basso a destra.

import os
from PIL import Image

--righe omesse--

    # Verifica se l'immagine deve essere ridimensionata.
--righe omesse--

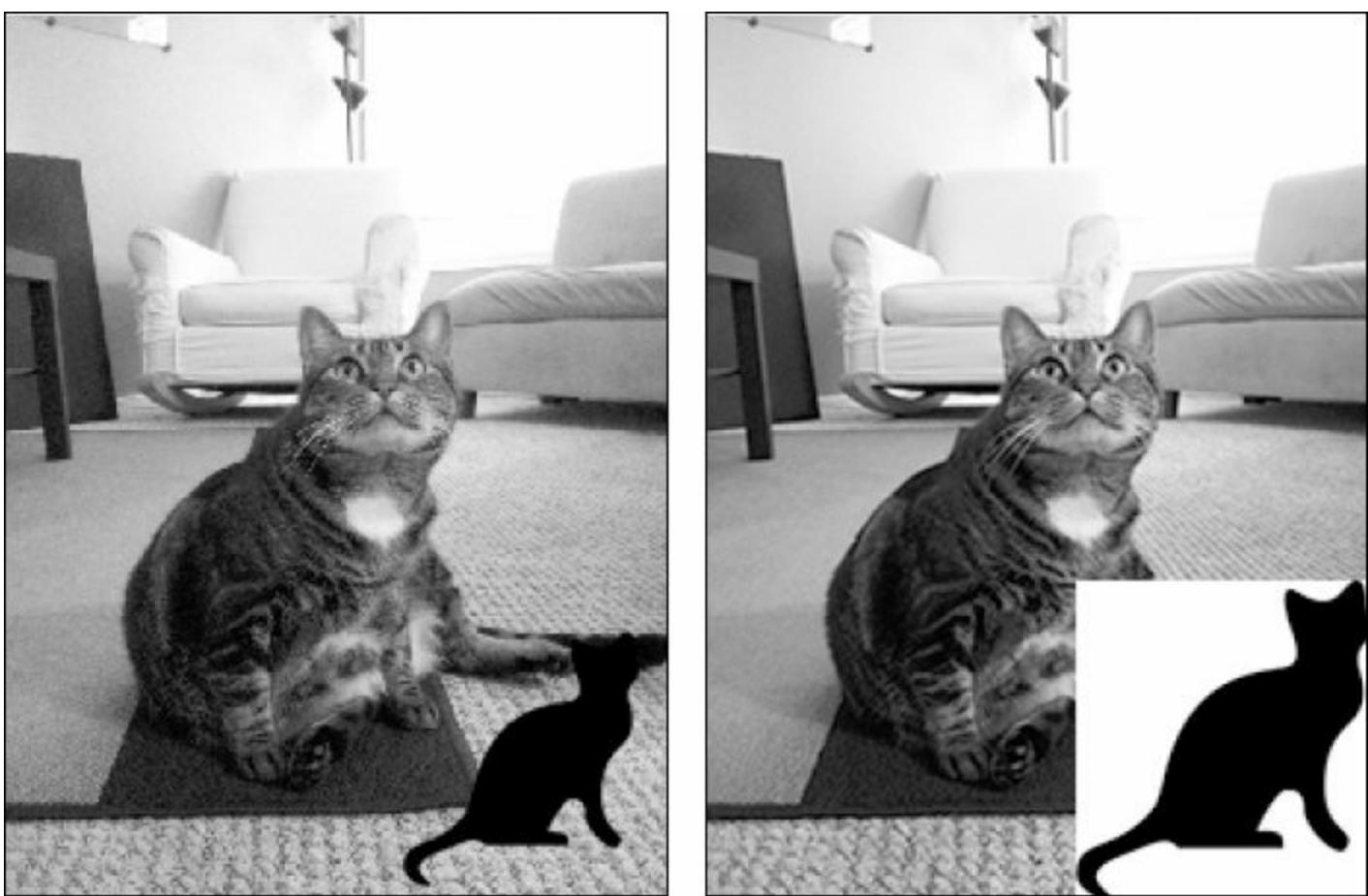
    # Aggiunge il logo.
❶    print('Sto aggiungendo il logo a %s...' % (filename))
❷    im.paste(logoIm, (width - logoWidth, height - logoHeight), logoIm)

    # Salva le modifiche.
❸    im.save(os.path.join('withLogo', filename))
```

Il nuovo codice stampa un messaggio in cui comunica all'utente che sta aggiungendo il logo ❶, incolla `logoIm` su `im` alle coordinate calcolate ❷, e salva le modifiche in un file nella directory `withLogo` ❸. Se eseguite questo programma con `zophie.png` come unica immagine nella directory di lavoro, l'output sarà:

```
Sto ridimensionando zophie.png...
Sto aggiungendo il logo a zophie.png...
```

L'immagine `zophie.png` verrà modificata in un'immagine di  $255 \times 300$  pixel che si presenta come nella Figura 17.13. ricordate che il metodo `paste()` non incollerà i pixel di trasparenza se non passate `logoIm` anche come terzo argomento. Questo programma può automaticamente ridimensionare e aggiungere un logo a centinaia di immagini in un paio di minuti soltanto.



**Figura 17.13** - L'immagine zophie.png ridimensionata con il logo aggiunto (a sinistra). Se dimenticate il terzo argomento, i pixel trasparenti nel logo verranno copiati come pixel bianchi (a destra).

## Idee per programmi simili

Saper comporre immagini o modificare le dimensioni delle immagini a lotti può essere utile in molte applicazioni. Potreste scrivere programmi simili che facciano le cose seguenti:

- aggiungere testo o l'URL di un sito web alle immagini;
- aggiungere timestamp alle immagini;
- copiare o spostare immagini in cartelle diverse a seconda delle loro dimensioni;
- aggiungere a un'immagine un watermark in gran parte trasparente per impedire che altri la copino.

## Disegnare su immagini

Se avete bisogno di disegnare linee, rettangoli, circonference o altre forme semplici su un'immagine, potete usare il modulo `ImageDraw` di Pillow. Inserite quanto segue nella shell interattiva:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
```

Come prima cosa, importiamo `Image` e `ImageDraw`. Poi creiamo una nuova immagine, in questo caso un'immagine bianca di  $200 \times 200$  pixel, e memorizziamo l'oggetto `Image` in `im`. Passiamo l'oggetto `Image` alla funzione `ImageDraw.Draw()` per ricevere un oggetto `ImageDraw`. Questo oggetto ha vari metodi per il disegno di forme e testo su un oggetto `Image`. Memorizziamo l'oggetto `ImageDraw` in una variabile `draw`,

in modo da poterlo usare facilmente nell'esempio che segue.

## Disegnare forme

I metodi di `ImageDraw` che vedremo disegnano vari tipi di forme sull'immagine. I parametri `fill` e `outline` per questi metodi sono facoltativi e hanno come valore predefinito il bianco se non vengono specificati.

### Punti

Il metodo `point(xy, fill)` disegna singoli pixel. L'argomento `xy` rappresenta una lista dei punti che si vogliono disegnare. La lista può essere una lista di tuple di coordinate `x` e `y`, come `[(x, y), (x, y), ...]`, oppure una lista di coordinate `x` e `y` senza tuple, come `[x1, y1, x2, y2, ...]`.

L'argomento `fill` è il colore dei punti ed è o una tupla RGBA o una stringa con il nome di un colore, per esempio '`red`'. L'argomento `fill` è facoltativo.

### Linee

Il metodo `line(xy, fill, width)` disegna una linea o una serie di linee. L'argomento `xy` è o una lista di tuple, come `[(x, y), (x, y), ...]`, oppure una lista di interi, come `[x1, y1, x2, y2, ...]`. Ciascun punto è uno degli estremi delle linee che si disegnano. L'argomento `fill`, facoltativo, è il colore delle linee, come tupla RGBA o come nome di colore. L'argomento facoltativo `width` è lo spessore delle linee ed è uguale a 1 come valore predefinito, se non viene specificato.

### Rettangoli

Il metodo `rectangle(xy, fill, outline)` disegna un rettangolo. L'argomento `xy` è una tupla di box, nella forma `(left, top, right, bottom)`. I valori `left` e `top` specificano le coordinate `x` e `y` del vertice superiore sinistro del rettangolo, mentre `right` e `bottom` specificano il vertice inferiore destro. L'argomento facoltativo `fill` è il colore di riempimento del rettangolo, mentre l'argomento facoltativo `outline` è il colore del bordo del rettangolo.

### Ellissi

Il metodo `ellipse(xy, fill, outline)` disegna un'ellisse. Se la larghezza e l'altezza dell'ellisse sono identiche, il metodo disegna un cerchio. L'argomento `xy` è una tupla di box `(left, top, right, bottom)` che rappresenta un riquadro in cui l'ellisse si inscrive esattamente. L'argomento facoltativo `fill` è il colore di riempimento dell'ellisse, l'argomento facoltativo `outline` è il colore del bordo dell'ellisse.

### Poligoni

Il metodo `polygon(xy, fill, outline)` disegna un poligono arbitrario. L'argomento `xy` è una lista di tuple, come `[(x, y), (x, y), ...]`, oppure una lista di interi, come `[x1, y1, x2, y2, ...]`, che rappresentano i vertici del poligono. L'ultima coppia di coordinate viene automaticamente collegata alla prima. L'argomento facoltativo `fill` è il colore di riempimento del poligono, l'argomento facoltativo `outline` è il colore del bordo del poligono.

## Un esempio di disegno

Inserite quanto segue nella shell interattiva:

```
>>> from PIL import Image, ImageDraw  
>>> im = Image.new('RGBA', (200, 200), 'white')  
>>> draw = ImageDraw.Draw(im)  
❶ >>> draw.line([(0, 0), (199, 0), (199, 199), (0, 199), (0, 0)], fill='black')  
❷ >>> draw.rectangle((20, 30, 60, 60), fill='blue')  
❸ >>> draw.ellipse((120, 30, 160, 60), fill='red')  
❹ >>> draw.polygon(((57, 87), (79, 62), (94, 85), (120, 90), (103, 113)), fill='brown')  
❺ >>> for i in range(100, 200, 10):  
    draw.line([(i, 0), (200, i - 100)], fill='green')  
  
>>> im.save('drawing.png')
```

Dopo aver creato un oggetto `Image` per un'immagine bianca di  $200 \times 200$  pixel, averlo passato a `ImageDraw.Draw()` per ottenere un oggetto `ImageDraw` e aver memorizzato l'oggetto `ImageDraw` in `draw`, potete chiamare i metodi di disegno su `draw`. Qui abbiamo creato un bordo sottile in nero, un rettangolo in blu con il vertice superiore sinistro in  $(20, 30)$  e il vertice inferiore destro in  $(60, 60)$  ❷, un'elisse in rosso definita da un box che va da  $(120, 30)$  a  $(160, 60)$  ❸, un poligono marrone di cinque lati ❹ e una serie di linee verdi disegnate con un ciclo `for` ❺. Il file `drawing.png` risultante sarà come nella Figura 17.14.

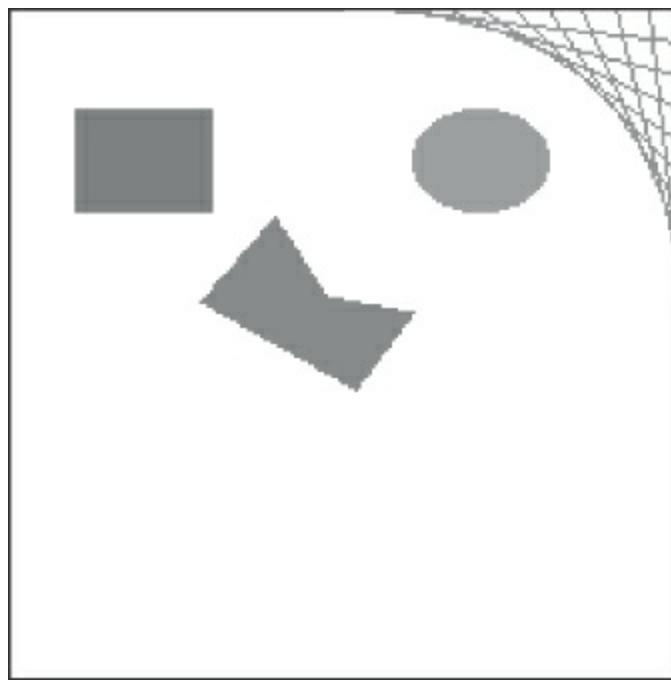


Figura 17.14 - L'immagine `drawing.png` risultante.

Esistono vari altri metodi di disegno di forme per oggetti `ImageDraw`. La documentazione completa si può consultare all'indirizzo <http://pillow.readthedocs.org/en/latest/reference/ImageDraw.html>.

## Disegnare testo

L'oggetto `ImageDraw` ha anche un metodo `text()` per disegnare del testo su un'immagine. Il metodo `text()` prende quattro argomenti: `xy`, `text`, `fill` e `font`.

- L'argomento `xy` è una tupla di due interi che specifica il vertice superiore sinistro del riquadro di testo.
- L'argomento `text` è la stringa di testo che si vuole scrivere.

- L'argomento facoltativo *fill* è il colore del testo.
- L'argomento facoltativo *font* è un oggetto `ImageFont`, utilizzato per stabilire il tipo di carattere e il corpo del testo. Lo analizzeremo meglio nella prossima sezione.

Poiché spesso è difficile sapere in anticipo quali saranno le dimensioni di un blocco di testo in un dato tipo di carattere, il modulo `ImageDraw` offre anche un metodo `textsize()`. Il suo primo argomento è la stringa di testo che si vuole misurare, il secondo argomento è un oggetto `ImageFont` facoltativo. Il metodo `textsize()` restituirà allora una tupla di due interi che rappresentano larghezza e altezza che avrebbe il testo nel tipo di carattere indicato, se venisse scritto sull'immagine. Potete usare quei valori di larghezza e altezza per calcolare esattamente dove volete inserire il testo sopra l'immagine. I primi tre argomenti di `text()` sono immediati. Prima di usare `text()` per disegnare del testo su un'immagine, vediamo il quarto argomento facoltativo, l'oggetto `ImageFont`.

Sia `text()` che `textsize()` prendono facoltativamente un oggetto `ImageFont` come ultimo argomento. Per creare uno di questi oggetti, bisogna prima eseguire:

```
>>> from PIL import ImageFont
```

Importato il modulo `ImageFont` di Pillow, potete chiamare la funzione `ImageFont.truetype()`, che prende due argomenti: il primo è una stringa per il file TrueType della font (è l'effettivo file della font che si trova sul vostro disco fisso). Un file TrueType ha l'estensione *.ttf* e di solito si trova in una di queste cartelle:

- in Windows: *C:\Windows\Fonts*
- in OS X: */Library/Fonts e /System/Library/Fonts*
- in Linux: */usr/share/fonts/truetype*

Non dovete inserire effettivamente questi percorsi nella stringa del file TrueType, perché Python sa di dover cercare automaticamente i tipi di caratteri in queste directory, ma visualizzerà un errore se non riesce a trovare la font che avete specificato.

Il secondo argomento per `ImageFont.truetype()` è un intero che indica la dimensione dei caratteri in punti (anziché, poniamo, in pixel). Ricordate che Pillow crea immagini PNG che per default sono a risoluzione di 72 pixel per pollice, e un punto è un 1/72-esimo di pollice.

Inserite quanto segue nella shell interattiva, sostituendo *FONT\_FOLDER* con il nome effettivo della cartella che utilizza il vostro sistema operativo:

```
>>> from PIL import Image, ImageDraw, ImageFont
>>> import os
❶ >>> im = Image.new('RGBA', (200, 200), 'white')
❷ >>> draw = ImageDraw.Draw(im)
❸ >>> draw.text((20, 150), 'Hello', fill='purple')
>>> fontsFolder = 'FONT_FOLDER' # per esempio '/Library/Fonts'
❹ >>> arialFont = ImageFont.truetype(os.path.join(fontsFolder, 'arial.ttf'), 32)
❺ >>> draw.text((100, 150), 'Howdy', fill='gray', font=arialFont)
>>> im.save('text.png')
```

Dopo aver importato `Image`, `ImageDraw`, `ImageFont` e `os`, creiamo un oggetto `Image` per una nuova imagine bianca di 200×200 pixel ❶ e creiamo un oggetto `ImageDraw` dall'oggetto `Image` ❷. Usiamo `text()` per

disegnare *Hello* alle coordinate (20, 150) in viola ❸. Non abbiamo passato il quarto argomento facoltativo in questa chiamata `text()`, perciò tipo di carattere e corpo non sono stati personalizzati. Per impostare tipo di carattere e corpo, prima memorizziamo il nome della cartella (per esempio */Library/Fonts*) in `fontsFolder`. Poi chiamiamo `ImageFont.truetype()`, passandogli il file *.ttf* per la font che vogliamo, seguito da un intero per il corpo ❹. Memorizziamo l'oggetto `Font` ottenuto da `ImageFont.truetype()` in una variabile come `arialFont`, poi passiamo la variabile a `text()` nell'ultimo argomento per parola chiave. La chiamata a `text()` in ❺ disegna *Howdy* nella posizione (100, 150) in grigio in Arial 32 punti.

Il file *text.png* risultante sarà simile a quello della [Figura 17.15](#).



**Figura 17.15** - L'immagine *text.png* risultante.

## Riepilogo

Le **immagini** sono costituite da una collezione di **pixel**, e ciascun pixel ha un valore **RGBA** per il **colore** ed è identificabile mediante **coordinate** *x* e *y*. Due formati molto diffusi per le immagini sono JPEG e PNG. Il modulo Pillow può gestire entrambi questi formati e anche altri.

Quando viene caricata un'immagine in un oggetto `Image`, la sua larghezza e la sua altezza sono memorizzate come una tupla di due interi nell'attributo `size`. Gli oggetti del tipo di dati `Image` hanno anche metodi per tipiche **manipolazioni** di immagini: `crop()`, `copy()`, `paste()`, `resize()`, `rotate()` e `transpose()`. Per salvare l'oggetto `Image` in un file di immagine, si chiama il metodo `save()`.

Se volete che il vostro programma disegni forme su un'immagine, usate i metodi di `ImageDraw` per **disegnare** punti, linee, rettangoli, ellissi e poligoni. Il modulo mette a disposizione anche metodi per disegnare testo in un tipo di carattere e in un corpo di vostra scelta.

Anche se applicazioni avanzate (e costose) come Photoshop mettono a disposizione capacità di elaborazione batch, potete usare script Python per effettuare molte delle stesse modifiche gratuitamente. Nei capitoli precedenti, avete scritto programmi Python per gestire file di puro testo, fogli di calcolo, PDF e altri formati. Con il modulo `pillow`, avete esteso le vostre capacità di programmazione all'elaborazione di immagini.

## Domande di ripasso

1. Che cos'è un valore RGBA?
2. Come si può ottenere il valore RGBA del colore 'CornflowerBlue' dal modulo Pillow?
3. Che cos'è una tupla di box?
4. Quale funzione restituisce un oggetto `Image` per, supponiamo, un file di immagine che si chiama *zophie.png*?

5. Come potete trovare larghezza e altezza dell'immagine di un oggetto `Image`?
6. Quale metodo chiamereste per ottenere un oggetto `Image` per un'immagine di  $100 \times 100$  pixel, escludendo il quadrante inferiore sinistro?
7. Dopo aver apportato delle modifiche a un oggetto `Image`, come potete salvarlo come file di immagine?
8. Quale modulo contiene il codice per il disegno di forme di `Pillow`?
9. Gli oggetti `Image` non hanno metodi di disegno. Quale tipo di oggetto li possiede? Come si ottiene questo tipo di oggetto?

## Un po' di pratica

Per esercitarvi, scrivete dei programmi che svolgano le attività seguenti.

## Estendere e sistemare i programmi del progetto di capitolo

Il programma `resizeAndAddLogo.py` di questo capitolo funziona con file PNG e JPEG, ma `Pillow` supporta molti altri formati. Estendetelo il programma in modo da elaborare anche le immagini GIF e BMP.

Un altro piccolo problema è che il programma modifica i file PNG e JPEG solo se le estensioni dei loro nomi di file sono in tutte minuscole. Per esempio, elaborerà `zophie.png` ma non `zophie.PNG`. Modificate il codice in modo che il controllo dell'estensione dei nomi di file non faccia distinzione fra maiuscole e minuscole.

Infine, il logo aggiunto nell'angolo inferiore destro vuole essere solo un piccolo segno, ma se l'immagine è all'incirca delle stesse dimensioni del logo stesso, il risultato sarà simile a quello della [Figura 17.16](#). Modificate `resizeAndAddLogo.py` in modo che verifichi che l'immagine sia almeno il doppio, in larghezza e altezza, dell'immagine del logo prima che il logo venga incollato; in caso contrario, deve evitare di aggiungere il logo.



**Figura 17.16** - Quando l'immagine non è molto più grande del logo, il risultato è sgradevole.

## Identificare le cartelle di foto sul disco fisso

Ho la cattiva abitudine di trasferire i file dalla fotocamera digitale in cartelle temporanee sul disco fisso, per poi dimenticarmi di dove le ho messe. Sarebbe bello scrivere un programma che possa esplorare tutto il disco fisso e trovare queste “cartelle di foto” abbandonate.

Scrivete un programma che esplori ogni cartella del vostro disco fisso e trovi le potenziali cartelle di foto. Ovviamente, dovete prima definire che cosa considerate una “cartella di foto”; supponiamo che sia qualsiasi cartella in cui oltre metà dei file sono fotografie. E come definite quali file sono foto? Innanzitutto, un file di foto deve avere l'estensione `.png` o `.jpg`. Inoltre, le foto sono immagini di grandi dimensioni: larghezza e altezza di una foto devono essere maggiori di 500 pixel. È una congettura plausibile, poiché la maggior parte delle fotocamere creano immagini di parecchie migliaia di pixel in larghezza e altezza.

Come suggerimento, ecco uno scheletro appena abbozzato di come potrebbe essere fatto questo programma:

```
#! python3
# Importa i moduli e scrive i commenti per descrivere il programma.

for foldername, subfolders, filenames in os.walk('C:\\\\'):
    numPhotoFiles = 0
    numNonPhotoFiles = 0
    for filename in filenames:
        # Verifica se l'estensione del file non è .png o .jpg.
        if DA FARE:
            numNonPhotoFiles += 1
            continue # salta al nome di file successivo
        # Apre il file di imagine con Pillow.
        # Verifica se larghezza e altezza sono maggiori di 500.
        if DA FARE:
            # L'immagine è abbastanza grande da poter essere considerate una foto.
            numPhotoFiles += 1
        else:
            # L'immagine è troppo piccola per essere una foto.
            numNonPhotoFiles += 1

    # Se oltre la metà dei file sono foto,
    # stampa il percorso assoluto della cartella.
    if DA FARE:
        print(DA FARE)
```

Quando il programma viene eseguito, deve stampare sullo schermo il percorso assoluto di tutte le cartelle di foto.

## Segnaposto personalizzati

Nel [Capitolo 13](#), uno dei progetti suggeriti come esercizio consisteva nel creare inviti personalizzati a partire da un elenco di invitati salvato in un file di puro testo. Come progetto ulteriore, usate il modulo `pillow` per creare immagini per i segnaposto per gli invitati. Per ciascun invitato elencato nel file `guests.txt` (che trovate fra le risorse a <http://nostarch.com/automatestuff/>), generate un file di immagine con il nome dell'ospite e qualche decorazione floreale. Un'immagine di fiore di pubblico dominio si trova nelle risorse a <http://nostarch.com/automatestuff/>.

Per essere sicuri che tutti i segnaposto siano delle stesse dimensioni, aggiungete un rettangolo nero ai

bordi dell'immagine dell'invito, in modo che, quando l'immagine verrà stampata, presenti una guida per il ritaglio. I file PNG che pillow produce sono impostati a 72 pixel per pollice, perciò un segnaposto da  $4 \times 5$  pollici (circa  $10 \times 12,5$  cm) richiederà un'immagine di  $288 \times 360$  pixel.

# Controllare tastiera e mouse con l'automazione della GUI

Conoscere vari moduli Python per modificare fogli di calcolo, scaricare file e lanciare programmi è utile, ma a volte semplicemente non ci sono moduli per le applicazioni con cui si deve lavorare. Gli strumenti ultimi per **automatizzare attività sul vostro computer** sono **programmi** scritti da voi che controllano direttamente tastiera e mouse.

Questi programmi possono controllare altre applicazioni inviando loro pressioni di tasti e clic di mouse virtuali, come se foste seduti davanti al computer e interagiste con le applicazioni in prima persona. Questa tecnica è chiamata **automazione dell'interfaccia grafica**, o **automazione GUI** per brevità. Con l'automazione GUI, i vostri programmi possono fare qualsiasi cosa possa fare un essere umano seduto davanti al computer, tranne rovesciare il caffè sulla tastiera.

Pensate l'automazione GUI come la programmazione di un braccio robotico. Potete programmare il braccio robotico perché scriva alla tastiera e sposti il mouse per voi. Questa tecnica è particolarmente utile per attività che richiedono una gran quantità di clic o di compilazione di moduli.

Il modulo `pyautogui` mette a disposizione funzioni per simulare movimenti del mouse, clic di pulsanti e scorrimento della rotellina. Questo capitolo esamina solo un sottoinsieme delle caratteristiche di PyAutoGUI; potete trovare la documentazione completa all'indirizzo <http://pyautogui.readthedocs.org/>.

## Installazione del modulo `pyautogui`

Il modulo `pyautogui` può inviare pressioni di tasti e clic del mouse virtuali a Windows, OS X e Linux. A seconda del sistema operativo che usate, dovrete forse installare ulteriori moduli (si chiamano

dipendenze) prima di poter installare PyAutoGUI.

- In Windows, non ci sono altri moduli da installare.
- In OS X, eseguite sudo pip3 install pyobjc-framework-Quartz, sudo pip3 install pyobjc-core, poi sudo pip3 install pyobjc.
- In Linux, eseguite sudo pip3 install python3-xlib, sudo apt-get install scrot, sudo apt-get install python3-tk, e infine sudo apt-get install python3-dev. (Scrot è un programma di cattura dello schermo utilizzato da PyAutoGUI.)

Installate queste dipendenze, eseguite pip install pyautogui (o pip3 sotto OS X e Linux) per installare PyAutoGUI. Nell'Appendice A trovate informazioni complete sull'installazione di moduli di terze parti. Per verificare se PyAutoGUI è stato installato correttamente, eseguite import pyautogui dalla shell interattiva: se non compaiono messaggi d'errore, l'installazione è andata a buon fine.

## Come rimanere sulla strada giusta

Prima di buttarvi a capofitto nell'automazione GUI, dovete sapere come sfuggire ai problemi che si possono presentare. Python può spostare il mouse e premere tasti a velocità incredibile; in effetti, potrebbe essere troppo veloce perché altri programmi riescano a tenere il suo passo. Inoltre, se qualcosa va storto ma il vostro programma continua a spostare il mouse sullo schermo, è difficile dire che cosa stia facendo o come risolvere il problema. Come la scopa magica dell'apprendista stregone in *Fantasia* di Disney, che continuava a riempire e a far traboccare d'acqua il secchio di Topolino, il vostro programma potrebbe sfuggire al controllo anche se sta seguendo le istruzioni alla perfezione. Fermare il programma può essere difficile, se il mouse si muove per lo schermo per i fatti suoi, impedendovi di fare un clic sulla finestra di IDLE per chiuderla. Per fortuna, però, esistono vari modi per impedire i problemi dell'automazione GUI o per recuperare il controllo se i problemi insorgono.

## Chiudere tutto con un logout

Forse il più semplice fra i modi per fermare un programma di automazione GUI fuori controllo è un **logout**, che chiude tutti i programmi in esecuzione. In Windows e Linux, la combinazione di tasti di logout è Ctrl-Alt-Canc; in OS X, è ⌘-Maiusc-Opzione-Q. Chiudendo tutto, perderete eventuale lavoro non salvato, ma per lo meno non dovete attendere un riavvio completo del computer.

## Pause e fail-safe

Potete dire al vostro script di attendere dopo ogni chiamata di funzione, presentandovi una finestra per riprendere il controllo del mouse e della tastiera se qualcosa va storto. Per farlo, impostate la variabile pyautogui.PAUSE al numero di secondi per cui volete che il programma resti in **pausa**.

Per esempio, impostando pyautogui.PAUSE = 1.5, ogni chiamata di funzione PyAutoGUI attenderà un secondo e mezzo dopo aver svolto la sua azione. Le istruzioni che non riguardano PyAutoGUI non avranno questa pausa.

PyAutoGUI ha anche una caratteristica **fail-safe**. Se si sposta il cursore nell'angolo superiore sinistro dello schermo, PyAutoGUI solleva un'eccezione pyautogui.FailSafeException. Il vostro programma può o gestire questa eccezione con enunciati try ed except, o lasciare che l'eccezione mandi in crash il

programma. In ogni caso, questa funzione di sicurezza fermerà il programma se spostate il mouse il più in alto e il più a sinistra possibile.

Potete disabilitare questa caratteristica impostando `pyautogui.FAILSAFE = False`. Inserite quanto segue nella shell interattiva:

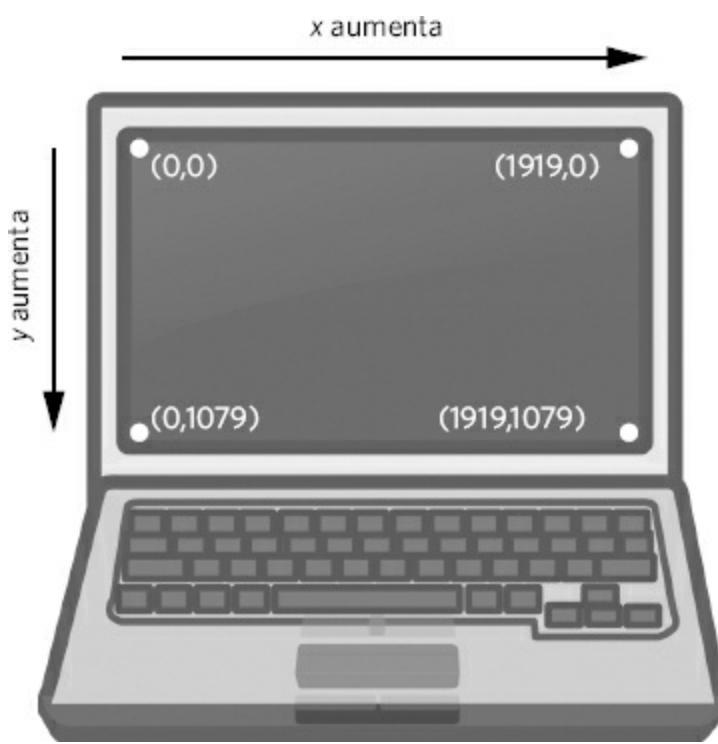
```
>>> import pyautogui  
>>> pyautogui.PAUSE = 1  
>>> pyautogui.FAILSAFE = True
```

Qui importiamo `pyautogui` e impostiamo `pyautogui.PAUSE` a 1 per una pausa di un secondo dopo ogni chiamata di funzione. Impostiamo `pyautogui.FAILSAFE` a `True` per abilitare questa funzionalità.

## Controllare i movimenti del mouse

In questa sezione, vedrete come spostare il mouse e tracciarne la posizione sullo schermo con PyAutoGUI, ma prima dovrete aver chiaro come PyAutoGUI lavori con le coordinate.

Le funzioni del mouse di PyAutoGUI usano coordinate  $x$  e  $y$ . La [Figura 18.1](#) mostra il sistema di coordinate per lo schermo del computer: è simile al sistema utilizzato per le immagini, che abbiamo visto nel [Capitolo 17](#). L'origine, in cui  $x$  e  $y$  sono entrambe 0, è nell'angolo superiore sinistro dello schermo. Le coordinate  $x$  aumentano andando verso destra, le coordinate  $y$  andando verso il basso. Tutte le coordinate sono interi positivi: non ci sono coordinate negative.



**Figura 18.1** - Le coordinate di uno schermo di computer con risoluzione 1920×1080.

La risoluzione è il numero di pixel dello schermo, in larghezza e altezza. Se la risoluzione del vostro schermo è impostata a 1920×1080, la coordinata dell'angolo superiore sinistro sarà (0, 0) e quella dell'angolo inferiore destro sarà (1919, 1079).

La funzione `pyautogui.size()` restituisce una tupla di due interi con la larghezza e l'altezza dello schermo in pixel. Inserite quanto segue nella shell interattiva:

```
>>> import pyautogui  
>>> pyautogui.size()  
(1920, 1080)  
>>> width, height = pyautogui.size()
```

pyautogui.size() restituisce (1920, 1080) su un computer con una risoluzione di  $1920 \times 1080$  pixel; a seconda della risoluzione del vostro schermo, il valore di ritorno potrebbe essere diverso. Potete memorizzare larghezza e altezza ricavate da pyautogui.size() in variabili come width e height, per una maggiore leggibilità dei vostri programmi.

## Spostare il mouse

Ora che conoscete le coordinate dello schermo, spostiamo il mouse. La funzione pyautogui.moveTo() sposterà immediatamente il cursore del mouse a una posizione specificata sullo schermo. Valori interi per le coordinate *x* e *y* costituiscono il primo e il secondo argomento della funzione, rispettivamente.

Un argomento per parola chiave facoltativo duration (intero o in virgola mobile) specifica il numero dei secondi che deve impiegare il mouse per arrivare alla sua destinazione. Se si omette questo argomento, il valore predefinito è 0, cioè il movimento è istantaneo. (Tutti gli argomenti per parola chiave duration nelle funzioni di PyAutoGUI sono facoltativi.)

Inserite quanto segue nella shell interattiva:

```
>>> import pyautogui  
>>> for i in range(10):  
    pyautogui.moveTo(100, 100, duration=0.25)  
    pyautogui.moveTo(200, 100, duration=0.25)  
    pyautogui.moveTo(200, 200, duration=0.25)  
    pyautogui.moveTo(100, 200, duration=0.25)
```

Questo esempio sposta il cursore del mouse in senso orario lungo un percorso quadrato fra le quattro coordinate, per dieci volte. Ogni movimento richiede un quarto di secondo, come specificato dall'argomento per parola chiave duration=0.25. Se non avessimo passato un terzo argomento alle chiamate pyautogui.moveTo(), il cursore del mouse sarebbe stato teletrasportato istantaneamente da punto a punto.

La funzione pyautogui.moveRel() sposta il cursore del mouse relativamente alla sua posizione corrente. L'esempio seguente sposta il mouse lungo lo stesso percorso quadrato, ma inizia il quadrato dal punto in cui si trova il mouse sullo schermo, qualunque esso sia, quando inizia l'esecuzione del codice:

```
>>> import pyautogui  
>>> for i in range(10):  
    pyautogui.moveRel(100, 0, duration=0.25)  
    pyautogui.moveRel(0, 100, duration=0.25)  
    pyautogui.moveRel(-100, 0, duration=0.25)  
    pyautogui.moveRel(0, -100, duration=0.25)
```

Anche pyautogui.moveRel() prende tre argomenti: di quanti pixel spostarsi orizzontalmente a destra, di quanti pixel spostarsi verticalmente verso il basso e (facoltativamente) quanto tempo deve richiedere il completamento del movimento. Un intero negativo per il primo o il secondo argomento farà sì che il mouse si sposti verso sinistra o verso l'alto, rispettivamente.

## Ottenere la posizione del mouse

Potete determinare la posizione del mouse chiamando la funzione `pyautogui.position()`, che restituisce una tupla con le posizioni  $x$  e  $y$  del cursore nel momento in cui la funzione è chiamata. Inserite quanto segue nella shell interattiva e spostate il mouse sullo schermo dopo ogni chiamata:

```
>>> pyautogui.position()
(311, 622)
>>> pyautogui.position()
(377, 481)
>>> pyautogui.position()
(1536, 637)
```

I valori che vi verranno restituiti saranno diversi, a seconda di dove si trova il cursore.

## Progetto: “Dov’è il mouse proprio adesso?”

Saper determinare la posizione del mouse è una parte importante dell’impostazione degli script di automazione GUI, ma è quasi impossibile stabilire le coordinate esatte di un pixel solo guardando lo schermo. Sarebbe comodo avere un programma che visualizzi costantemente le coordinate  $x$  e  $y$  del cursore del mouse mentre lo si sposta.

Ad alto livello, ecco che cosa deve fare il programma:

- visualizzare le coordinate  $x$  e  $y$  correnti del cursore del mouse;
- aggiornate quelle coordinate mentre il mouse si sposta sullo schermo.

Questo significa che il codice dovrà fare le cose seguenti:

- chiamare la funzione `position()` per recuperare le coordinate correnti;
- cancellare le coordinate stampate in precedenza stampando caratteri `\b` (backspace) sullo schermo;
- gestire l’eccezione `KeyboardInterrupt` in modo che l’utente possa premere Ctrl-C per uscire.

Aprite una nuova finestra di file editor e salvate il file con il nome `mouseNow.py`.

## Passo 1: importare il modulo

Iniziate il programma così:

```
#! python3
# mouseNow.py – Visualizza la posizione corrente del cursore del mouse.
import pyautogui
print('Premi Ctrl-C per uscire.')
# DA FARE: Ottenere e stampare le coordinate del mouse.
```

L’inizio del programma importa il modulo `pyautogui` e stampa quindi un promemoria per l’utente, ricordandogli che deve premere Ctrl-C per uscire.

## Passo 2: impostare il codice di uscita e il ciclo infinito

Potete usare un ciclo while infinito per stampare costantemente le coordinate correnti del mouse da `mouse.position()`. Per quanto riguarda il codice che esce dal programma, dovrete catturare l'eccezione `KeyboardInterrupt`, che viene sollevata ogni qualvolta l'utente preme Ctrl-C. Se non gestite questa eccezione, visualizzerà all'utente un terribile messaggio di traceback e di errore. Aggiungete quanto segue al vostro programma:

```
#! python3
# mouseNow.py - Visualizza la posizione corrente del cursore del mouse.
import pyautogui
print('Premi Ctrl-C per uscire.')
try:
    while True:
        # DA FARE: Ottenere e stampare le coordinate del mouse.
❶    except KeyboardInterrupt:
❷    print('\nFatto.')
```

Per gestire l'eccezione, racchiudete il ciclo `while` infinito in un enunciato `try`. Quando l'utente preme Ctrl-C, l'esecuzione del programma passerà alla clausola `except` ❶ e su una nuova riga verrà stampato Fatto. ❷.

### Passo 3: ottenere e stampare le coordinate del mouse

Il codice all'interno del ciclo `while` deve ottenere le coordinate correnti del mouse, formellarle in modo elegante e stamparle. Aggiungete il codice seguente all'interno del ciclo `while`:

```
#! python3
# mouseNow.py - Visualizza la posizione corrente del cursore del mouse.
import pyautogui
print('Premi Ctrl-C per uscire.')
--righe omesse--
    # Ottiene e stampa le coordinate del mouse.
    x, y = pyautogui.position()
    positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)
-- righe omesse --
```

Utilizzando il trucco dell'assegnazione multipla, alle variabili `x` e `y` vengono dati i valori dei due interi restituiti nella tupla da `pyautogui.position()`. Passando `x` e `y` alla funzione `str()`, potete ottenere le coordinate intere sotto forma di stringa. Il metodo stringa `rjust()` allineerà a destra i valori, in modo che possano prendere la stessa quantità di spazio, indipendentemente dal fatto che la coordinata abbia una, due, tre o quattro cifre. Concatenando le coordinate stringa allineate a destra con etichette 'X:' e 'Y:' si ottiene una stringa ben formattata, che verrà memorizzata in `positionStr`.

Alla fine del programma, aggiungete il codice seguente:

```
#! python3
# mouseNow.py - Visualizza la posizione corrente del cursore del mouse.
--righe omesse--
    print(positionStr, end='')
❶    print('\b' * len(positionStr), end='', flush=True)
```

Questo stampa effettivamente `positionStr` sullo schermo. L'argomento per parola chiave `end=""` di `print()` impedisce che alla fine della riga stampata venga aggiunto il carattere newline di default. È possibile

cancellare testo che è già stato stampato sullo schermo – ma solo per l’ultima riga di testo. Una volta che si stampa un carattere newline, non si può cancellare nulla che sia stato stampato prima di quel carattere.

Per cancellare testo, stampate il carattere escape \b di backspace. Questo carattere speciale cancella un carattere alla fine della riga corrente sullo schermo. La riga in ❶ usa la replica di stringhe per produrre una stringa con tanti caratteri \b quanto è lunga la stringa memorizzata in positionStr, il che ha l’effetto di cancellare l’ultima stringa positionStr che è stata stampata.

Per ragioni tecniche che vanno al di là delle finalità di questo libro, passate sempre flush=True alle chiamate print() che stampano caratteri backspace \b. Altrimenti, lo schermo potrebbe non aggiornare il testo come desiderato.

Dato che il ciclo while si ripete così rapidamente, l’utente in effetti non si renderà conto che il numero intero sullo schermo viene cancellato e ristampato. Per esempio, se la coordinata *x* è 563 e il mouse si sposta di un pixel a destra, sembrerà che solo il 3 in 563 cambi in un 4.

Quando eseguite il programma, verranno stampate solo due righe. Saranno simili a queste:

```
Premi Ctrl-C per uscire.  
X: 290 Y: 424
```

La prima riga visualizza l’istruzione: bisogna premere Ctrl-C per uscire. La seconda con le coordinate del mouse si modificherà mentre spostate il mouse sullo schermo. Con questo programma, potrete stabilire le coordinate del mouse per i vostri script di automazione GUI.

## Controllare le interazioni del mouse

Ora che sapete come spostare il mouse e stabilire dove si trova sullo schermo, siete pronti per iniziare a **fare clic, a trascinare e a scorrere**.

### Fare clic con il mouse

Per inviare un clic virtuale del mouse al computer, chiamate il metodo pyautogui.click(). Per default questo clic usa il pulsante sinistro del mouse e avviene ovunque si trovi il cursore del mouse in quel momento. Potete passare le coordinate *x* e *y* del clic come primo e secondo argomento facoltativi, se volete che il clic avvenga in qualche posizione diversa da quella corrente del mouse.

Se volete specificare quale pulsante del mouse usare, includete l’argomento per parola chiave button, con valore 'left', 'middle', o 'right'.

Per esempio, pyautogui.click(100, 150, button='left') effettuerà un clic del pulsante sinistro del mouse alle coordinate (100, 150), mentre pyautogui.click(200, 250, button='right') farà un clic destro nella posizione (200, 250).

Inserite quanto segue nella shell interattiva:

```
>>> import pyautogui  
>>> pyautogui.click(10, 5)
```

Dovreste vedere il puntatore del mouse che si sposta vicino all’angolo superiore sinistro dello schermo e fa clic una volta. Un “clic” completo è definito come una pressione del mouse, con il successivo rilascio senza spostamento del cursore. Potete eseguire un clic anche chiamando pyautogui.mouseDown(), che determina solo la pressione del pulsante del mouse, e pyautogui.mouseUp(), che determina solo il rilascio del pulsante. Queste funzioni hanno gli stessi argomenti di click() e, in effetti,

la funzione `click()` non è altro che un comodo contenitore per queste due chiamate di funzione. Come ulteriore comodità, la funzione `pyautogui.doubleClick()` esegue due clic con il pulsante sinistro del mouse, mentre le funzioni `pyautogui.rightClick()` e `pyautogui.middleClick()` eseguono un clic rispettivamente con i pulsanti del mouse destro e centrale.

## Trascinare il mouse

Trascinare (*drag*) significa spostare il mouse mentre si tiene premuto uno dei pulsanti. Per esempio, si possono spostare file da una cartella all'altra trascinando le icone, oppure si possono spostare gli appuntamenti in una app di calendario.

PyAutoGUI mette a disposizione le funzioni `pyautogui.dragTo()` e `pyautogui.dragRel()` per trascinare il cursore del mouse in un nuova posizione assoluta o in una posizione relativamente a quella corrente. Gli argomenti per `dragTo()` e `dragRel()` sono gli stessi che per `moveTo()` e `moveRel()`: la coordinata *x*/spostamento orizzontale, la coordinata *y*/spostamento verticale, e facoltativamente una durata. (OS X non effettua trascinamenti in modo corretto, quando il mouse si sposta troppo rapidamente, perciò è consigliabile passare un argomento per parola chiave `duration`.)

Per mettere alla prova queste funzioni, aprete un'applicazione di disegno come Paint in Windows, Paintbrush in OS X o GNU Paint in Linux. (Se non avete un'applicazione di disegno, potete usarne una online, per esempio <http://sumopaint.com/>.) Useremo PyAutoGUI per disegnare in queste applicazioni.

Con il cursore del mouse posto sopra il canvas dell'applicazione di disegno, selezionato lo strumento matita o pennello, inserite quanto segue in una nuova finestra di file editor e salvate il file con il nome *spiralDraw.py*:

```
import pyautogui, time
❶ time.sleep(5)
❷ pyautogui.click() # fa clic per mettere il focus sul programma di disegno
distance = 200
while distance > 0:
    ❸     pyautogui.dragRel(distance, 0, duration=0.2) # sposta a destra
    ❹     distance = distance - 5
    ❺     pyautogui.dragRel(0, distance, duration=0.2) # sposta giù
    ❻     pyautogui.dragRel(-distance, 0, duration=0.2) # sposta a sinistra
    ❼     distance = distance - 5
    ❽     pyautogui.dragRel(0, -distance, duration=0.2) # sposta su
```

Quando si esegue questo programma, ci sarà un ritardo di cinque secondi ❶ per permettervi di spostare il cursore del mouse sulla finestra del programma di disegno, con lo strumento matita o pennello selezionato. Poi *spiralDraw.py* assumerà il controllo del mouse e farà clic per portare il **focus** sul programma di disegno ❷. Una finestra ha il focus quando possiede un cursore lampeggiante attivo, e le azioni che intraprendete (come scrivere o, in questo caso, trascinare il mouse) influenzano quella finestra. Una volta che il focus è sul programma di disegno, *spiralDraw.py* disegna una spirale quadrata come quella della [Figura 18.2](#).

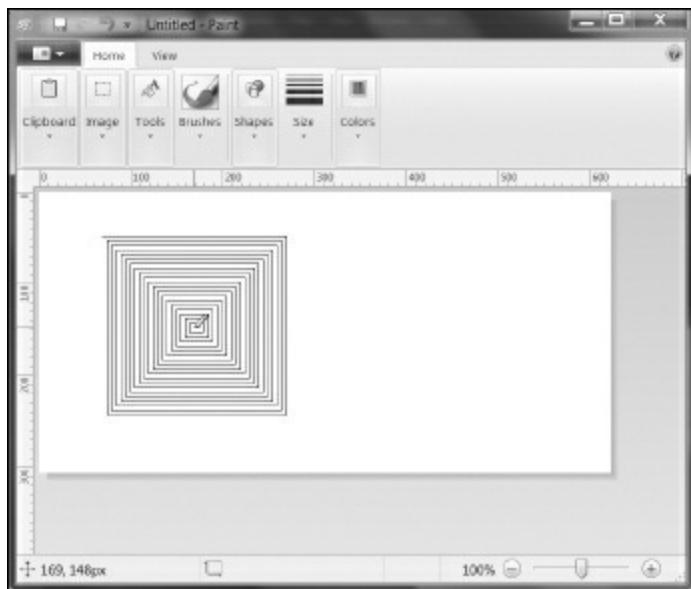


Figura 18.2 - I risultati dell'esempio pyautogui.dragRel().

La variabile `distance` inizia a 200, perciò alla prima iterazione del ciclo `while` la prima chiamata `dragRel()` trascina il cursore 200 pixel a destra, impiegando 0.2 secondi ❸. Poi `distance` viene diminuita a 195 ❹ e la seconda chiamata `dragRel()` trascina il cursore 195 pixel verso il basso ❺. La terza chiamata `dragRel()` trascina il cursore -195 in orizzontale (195 a sinistra) ❻, `distance` viene diminuita a 190, e l'ultima chiamata `dragRel()` trascina il cursore 190 pixel verso l'alto. A ogni iterazione, il mouse viene trascinato a destra, verso il basso, a sinistra e verso l'alto e `distance` è leggermente minore di quanto fosse nell'iterazione precedendo. Iterando su questo codice, potete spostare il cursore del mouse per disegnare una spirale quadrata.

Potreste disegnare questa spirale a mano (o meglio, con il mouse), ma dovreste lavorare molto lentamente per essere così precisi. PyAutoGUI può farlo in pochi secondi.

## NOTA

Potreste far disegnare l'immagine al codice con le funzioni di disegno del modulo `pillow`: vedete il [Capitolo 17](#) per maggiori informazioni. Utilizzando l'automazione GUI però potete usare gli strumenti di disegno avanzati che mettono a disposizione i programmi di grafica, come i gradienti, pennelli diversi o il secchiello.

## Scorrere il mouse

L'ultima funzione di PyAutoGUI per il mouse è `scroll()`, a cui si può passare come argomento un numero intero che indica di quante unità si vuole far scorrere il mouse verso l'alto o verso il basso. Che cosa sia “una unità” dipende dal sistema operativo e dall'applicazione, perciò dovete fare qualche esperimento per vedere esattamente l'entità dello scorrimento nelle vostre particolari condizioni. Lo scorrimento parte dalla posizione attuale del cursore del mouse. Se l'intero passato come argomento è positivo, lo scorrimento è verso l'alto, se è negativo lo scorrimento è verso il basso. Inserite quanto segue nella shell interattiva mentre il cursore si trova sopra la finestra di IDLE:

```
>>> pyautogui.scroll(200)
```

Vedrete IDLE scorrere brevemente verso l'alto – e poi ridiscendere. Lo scorrimento verso il basso si verifica perché IDLE scorre automaticamente verso il fondo dopo l'esecuzione di un'istruzione. Inserite invece questo codice:

```
>>> import pyperclip  
>>> numbers = ""  
>>> for i in range(200):  
    numbers = numbers + str(i) + '\n'  
  
>>> pyperclip.copy(numbers)
```

Questo importa pyperclip e imposta una stringa vuota, numbers. Il codice poi itera su 200 numeri e aggiunge ciascun numero a numbers, insieme con un carattere di newline. Dopo pyperclip.copy(numbers), negli Appunti si troveranno 200 righe di numeri.

Aprite una nuova finestra di file editor e incollateci il testo. Questo vi darà una lungafinestra di testo per mettere alla prova lo scorrimento. Inserite quanto segue nella shell interattiva:

```
>>> import time, pyautogui  
>>> time.sleep(5); pyautogui.scroll(100)
```

Nella seconda riga abbiamo inserito due comandi separati da un segno di punto e virgola, che dice a Python di eseguire i comandi come se fossero su righe separate. L'unica differenza è che la shell interattiva non presenterà il suo prompt per l'inserimento fra le due istruzioni. Questo è importante per l'esempio, perché vogliamo che la chiamata a pyautogui avvenga automaticamente dopo l'attesa. (Notate che avere due comandi su una sola riga può essere utile nella shell interattiva, ma nei programmi bisogna che ciascuna istruzione sia su una riga distinta.)

Dopo aver premuto Invio per mandare in esecuzione il codice, avrete cinque secondi per fare clic sulla finestra del file editor e attribuirle il focus. Una volta trascorsa la pausa, la chiamata pyautogui.scroll() farà sì che la finestra del file editor scorra verso l'alto dopo il ritardo di cinque secondi.

## Lavorare con lo schermo

I programmi di automazione GUI non devono per forza fare clic e scrivere alla cieca. PyAutoGui ha funzioni di **cattura dello schermo (screenshot)** che possono creare un file di immagine basato sui contenuti correnti dello schermo. Queste funzioni possono anche restituire un oggetto Image di pillow con l'aspetto che ha lo schermo in quel momento. Potete eventualmente leggere il [Capitolo 17](#) e installare il modulo pillow prima di continuare.

Su una macchina Linux, per usare le funzioni di screenshot di PyAutoGUI bisogna che sia installato il programma scrot. In una finestra di Terminale, eseguite sudo apt-get install scrot per installare questo programma. Se avete un computer con Windows oppure OS X, potete saltare questo passaggio e andare avanti.

## Catturare una schermata

Per catturare schermate in Python, chiamate la funzione pyautogui.screenshot(). Inserite quanto segue nella shell interattiva:

```
>>> import pyautogui  
>>> im = pyautogui.screenshot()
```

La variabile `im` conterrà l'oggetto `Image` della schermata. Ora potete chiamare dei metodi sull'oggetto `Image` nella variabile `im`, come per qualsiasi altro oggetto `Image`. Inserite quanto segue nella shell interattiva:

```
>>> im.getpixel((0, 0))  
(176, 176, 175)  
>>> im.getpixel((50, 200))  
(130, 135, 144)
```

Passate a `getpixel()` una tupla di coordinate, come `(0, 0)` o `(50, 200)`, e vi dirà il colore del pixel che ha quelle coordinate nell'immagine. Il valore restituito da `getpixel()` è una tupla RGB formata da tre interi che danno la quantità rispettivamente di rosso, verde e blu nel pixel. (Non c'è un quarto valore per alfa, perché gli screenshot sono completamente opachi.)

È questo il modo in cui i vostri programmi possono “vedere” quello che si trova sullo schermo.

## Analizzare la schermata

Supponiamo che uno dei passi nel vostro programma di automazione GUI consista nel fare clic su un pulsante grigio. Prima di chiamare il metodo `click()`, potrete catturare una schermata e cercare il pixel su cui lo script deve fare clic. Se non è dello stesso grigio del pulsante grigio, il programma sa che qualcosa non va. Forse la finestra si è spostata in maniera imprevista, o magari una finestra di dialogo a comparsa ha bloccato il pulsante. A questo punto, anziché continuare (e magari fare un disastro con un clic sulla cosa sbagliata) il programma può “vedere” che non farebbe clic nel posto giusto e si fermerebbe.

La funzione `pixelMatchesColor()` di PyAutoGUI restituirà `True` se il pixel alle coordinate `x` e `y` date sullo schermo corrisponde al colore dato. I primi due argomenti sono interi per le coordinate `x` e `y`, il terzo argomento è una tupla di tre interi per il colore RGB a cui deve corrispondere il pixel sullo schermo. Inserite quanto segue nella shell interattiva:

```
>>> import pyautogui  
>>> im = pyautogui.screenshot()  
❶ >>> im.getpixel((50, 200))  
(130, 135, 144)  
❷ >>> pyautogui.pixelMatchesColor(50, 200, (130, 135, 144))  
True  
❸ >>> pyautogui.pixelMatchesColor(50, 200, (255, 135, 144))  
False
```

Dopo aver catturato uno screenshot e aver usato `getpixel()` per ottenere una tupla RGB per il colore di un pixel alle coordinate specificate ❶, passate le stesse coordinate e la tupla RGB a `pixelMatchesColor()` ❷, che dovrebbe restituire `True`. Poi modificate un valore nella tupla RGB e chiamate nuovamente `pixelMatchesColor()` per le stesse coordinate ❸. Questo dovrebbe restituire `False`.

Questo metodo può essere utile ogni volta che i programmi di automazione GUI stanno per chiamare `click()`.

Notate che il colore alle coordinate date deve corrispondere esattamente. Se è anche leggermente diverso, per esempio `(255, 255, 254)` invece di `(255, 255, 255)`, `pixelMatchesColor()` restituirà `False`.

## Progetto: estendere il programma mouseNow

Potete estendere il progetto *mouseNow.py* affrontato in precedenza in questo capitolo, in modo che dia non solo le coordinate *x* e *y* della posizione corrente del cursore del mouse, ma anche il colore RGB del pixel che si trova sotto il cursore. Modificate il codice all'interno del ciclo while di *MouseNow.py* in questo modo:

```
#! python3
# mouseNow.py - Visualizza la posizione corrente del mouse.
--righe omesse--
    positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)
    pixelColor = pyautogui.screenshot().getpixel((x, y))
    positionStr += ' RGB: (' + str(pixelColor[0]).rjust(3)
    positionStr += ', ' + str(pixelColor[1]).rjust(3)
    positionStr += ', ' + str(pixelColor[2]).rjust(3) + ')'
    print(positionStr, end='')
--righe omesse--
```

Ora, quando eseguite *mouseNow.py*, l'output includerà anche il valore di colore RGB del pixel su cui si trova il cursore del mouse.

```
Premi Ctrl-C per uscire.
X: 406 Y: 17 RGB: (161, 50, 50)
```

Questa informazione, insieme con la funzione `pixelMatchesColor()`, faciliterà l'aggiunta di controlli del colore dei pixel ai vostri script di automazione GUI.

## Riconoscimento di immagini

Ma che cosa si può fare, se non si sa in anticipo dove debba fare clic PyAutoGUI? Potete usare il **riconoscimento di immagini**. Date a PyAutoGUI un'immagine dell'oggetto su cui volette fare clic e lasciate che determini da sé le coordinate.

Per esempio, se avete catturato uno screenshot per avere l'immagine di un pulsante *Submit (Invia)*, salvandola come *submit.png*, la funzione `locateOnScreen()` restituirà le coordinate alle quali si trova quell'immagine. Per vedere come funziona `locateOnScreen()`, provate a catturare uno screenshot di una piccola area dello schermo; poi salvate l'immagine e inserite quanto segue nella shell interattiva, sostituendo a '*submit.png*' il nome di file del vostro screenshot:

```
>>> import pyautogui
>>> pyautogui.locateOnScreen('submit.png')
(643, 745, 70, 29)
```

La tupla di quattro interi restituita da `locateOnScreen()` ha la coordinata *x* del bordo sinistro, la coordinata *y* del bordo superiore, la larghezza e l'altezza per la prima posizione sullo schermo in cui è stata trovata l'immagine. Se sperimentate questo procedimento sul vostro computer con una vostra cattura dello schermo, il valore di ritorno sarà diverso da quello indicato qui.

Se l'immagine non viene trovata sullo schermo, `locateOnScreen()` restituirà `None`. Notate che l'immagine sullo schermo deve corrispondere esattamente all'immagine fornita, per poter essere riconosciuta. Se l'immagine è diversa anche per un solo pixel, `locateOnScreen()` restituirà `None`.

Se l'immagine può essere trovata in più punti dello schermo, `locateAllOnScreen()` restituirà un oggetto

Generator, che può essere passato a `list()` per avere di ritorno una lista di tuple di quattro interi, una tupla per ciascuna posizione sullo schermo in cui l'immagine è stata identificata. Continuate l'esempio nella shell interattiva inserendo quanto segue (e sostituendo a 'submit.png' il nome di file della vostra immagine):

```
>>> list(pyautogui.locateAllOnScreen('submit.png'))  
[(643, 745, 70, 29), (1007, 801, 70, 29)]
```

Ciascuna delle tuple di quattro interi rappresenta un'area sullo schermo. Se la vostra immagine si trova in una sola posizione, `list()` e `locateAllOnScreen()` restituiranno una lista con una sola tupla. Una volta che avete la tupla di quattro interi per l'area dello schermo in cui è stata trovata la vostra immagine, potete fare clic al centro di quell'area passando la tupla alla funzione `center()` per avere di ritorno le coordinate `x` e `y` del centro di quell'area. Inserite quanto segue nella shell interattiva, sostituendo agli argomenti i vostri valori per nome di file, tupla di quattro interi e coppia di coordinate:

```
>>> pyautogui.locateOnScreen('submit.png')  
(643, 745, 70, 29)  
>>> pyautogui.center((643, 745, 70, 29))  
(678, 759)  
>>> pyautogui.click((678, 759))
```

Quando avete le coordinate del centro da `center()`, passandole a `click()` verrà eseguito un clic al centro dell'area dello schermo che corrisponde all'immagine che avete passato a `locateOnScreen()`.

## Controllare la tastiera

PyAutoGUI ha funzioni anche per l'invio di pressioni di **tasti virtuali** al computer, il consente di compilare moduli o di inserire del testo nelle applicazioni.

## Inviare una stringa dalla tastiera

La funzione `pyautogui.typewrite()` invia pressioni di tasti virtuali al computer. Quel che fanno questi tasti virtuali dipende da quale finestra e quale campo di testo abbiano il focus. Potete inviare prima un clic del mouse al campo di testo desiderato, per essere sicuri che abbia il focus.

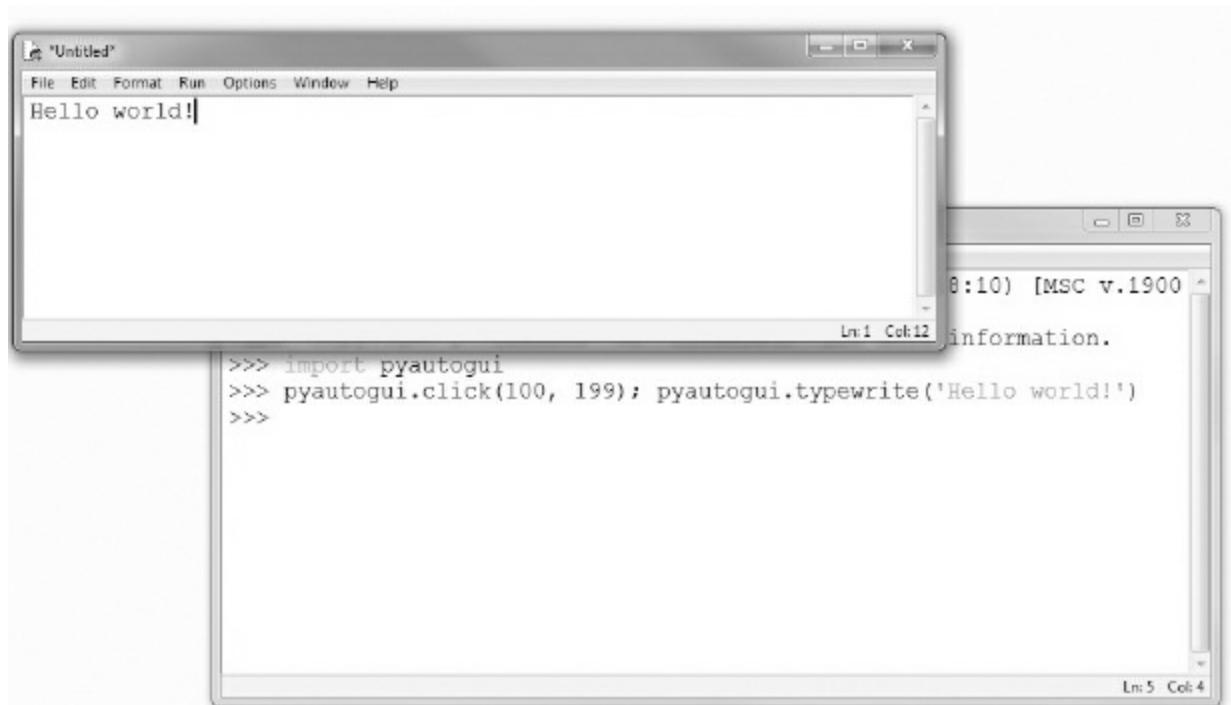
Come semplice esempio, usiamo Python per scrivere automaticamente le parole *Hello world!* in una finestra di file editor. Aprite innanzitutto una nuova finestra di file editor e collocatela nell'angolo superiore sinistro del vostro schermo, in modo che PyAutoGUI faccia clic nella posizione giusta per renderla attiva. Poi, inserite quanto segue nella shell interattiva:

```
>>> pyautogui.click(100, 100); pyautogui.typewrite('Hello world!')
```

Notate come collocare due comandi sulla stessa riga, separati da un segno di due punti, impedisca alla shell interattiva di emettere un novo prompt per invitarvi a un nuovo inserimento fra l'esecuzione di una istruzione e l'altra. Questo evita che possiate accidentalmente attivare una nuova finestra fra le chiamate `click()` e `typewrite()`, il che creerebbe un pasticcio.

Python prima invierà un clic virtuale del mouse alle coordinate (100, 100), che dovrebbe essere un clic sulla finestra del file editor, per attribuirle il focus. Poi la chiamata `typewrite()` invierà alla finestra il

testo *Hello world!*, così che il risultato sia analogo a quello visibile nella [Figura 18.3](#). Ora avete del codice che può scrivere per voi!



**Figura 18.3** - Uso di PyAutoGUI per fare un clic nella finestra del file editor e scrivere al suo interno la frase Hello world!.

Per default, la funzione `typewrite()` scriverà istantaneamente tutta la stringa. Potete però passare un secondo argomento facoltativo per aggiungere una breve pausa fra un carattere e l'altro. Questo secondo argomento è un valore intero o in virgola mobile con il numero dei secondi di pausa. Per esempio, `pyautogui.typewrite('Hello world!', 0.25)` attenderà un quarto di secondo dopo aver scritto *H*, un altro quarto di secondo dopo la *e*, e così via. Questo “effetto macchina per scrivere” può essere utile per applicazioni più lente, che non sono in grado di elaborare i segnali dei tasti abbastanza rapidamente da tenere il passo con PyAutoGUI.

Per caratteri come *H* o *!*, PyAutoGUI simula automaticamente anche la pressione del tasto delle maiuscole.

## Nomi dei tasti

Non tutti i tasti sono facilmente rappresentabili con singoli caratteri di testo. Per esempio, come si rappresentano Maiusc o il tasto freccia a sinistra come caratteri singoli? In PyAutoGUI, questi tasti sono rappresentati da brevi valori stringa, come `'esc'` for il tasto Esc key o `'enter'` per il tasto Invio.

Anziché un singolo argomento stringa, si può passare a `typewrite()` una lista di queste stringhe di tasto. Per esempio, la chiamata seguente preme il tasto *a*, poi il tasto *b*, poi due volte la freccia a sinistra e infine i tasti *x* e *y*:

```
>>> pyautogui.typewrite(['a', 'b', 'left', 'left', 'X', 'Y'])
```

Poiché la pressione del tasto freccia sinistra sposta il cursore, il risultato finale sarà *xyab*. La [Tabella 18.1](#) (nella pagina seguente) elenca le stringhe per i tasti che si possono passare a `typewrite()` per simulare la pressione di qualsiasi combinazione da tastiera.

**Tabella 18.1** - Le stringhe che identificano i tasti in PyKeyboard.

Stringhe dei tasti	Significato
'a', 'b', 'c', 'A', 'B', 'C', '1', '2', '3', '!', '@', '#' ecc.	I tasti dei caratteri singoli
'enter' ( o 'return' o '\n')	Il tasto Invio
'esc'	Il tasto Esc
'shiftleft', 'shiftright'	I tasti Maiusc sinistro e destro
'altright', 'altright'	I tasti Alt sinistro e destro
'ctrlleft', 'ctrlright'	I tasti Ctrl sinistro e destro
'tab' ( o 't')	Il tasto Tab
'backspace', 'delete'	I tasti backspace e Canc
'pageup', 'pagedown'	I tasti PGSU e PGGIÙ
'home', 'end'	I tasti Home e Fine
'up', 'down', 'left', 'right'	I tasti freccia su, giù, sinistra, destra
'f1', 'f2', 'f3' ecc.	I tasti da F1 a F12
'volumemute', 'volumedown', 'volumeup'	I tasti mute, abbassa il volume e alza il volume (alcune tastiere non hanno questi tasti, ma il sistema operativo sarà comunque in grado di comprendere questi tasti virtuali)
'pause'	Il tasto Pausa
'capslock', 'numlock', 'scrolllock'	I tasti Bloc Maiusc, Bloc Num e Bloc Scorr
'insert'	Il tasto Ins o Insert
'printscreen'	Il tasto Stamp o Print Screen
'winleft', 'winright'	I tasti Win sinistro e destro (in Windows)
'command'	Il tasto Comando (⌘) (in OS X)
'option'	Il tasto Opzione (in OS X)

Potete anche esaminare la lista `pyautogui.KEYBOARD_KEYS` per vedere tutte le possibili stringhe di tasti che PyAutoGUI accetta. La stringa 'shift' si riferisce al tasto Maiusc di sinistra ed equivale a 'shiftleft'. Lo stesso vale per le stringhe 'ctrl', 'alt' e 'win': si riferiscono tutte al tasto di sinistra.

## Premere e rilasciare la tastiera

Come le funzioni `mouseDown()` e `mouseUp()`, `pyautogui.keyDown()` e `pyautogui.keyUp()` invieranno al computer la pressione e il rilascio di tasti virtuali. A queste funzioni vengono passate come argomento le stringhe dei tasti ([Tabella 18.1](#)). Per comodità, PyAutoGUI mette a disposizione una funzione `pyautogui.press()`, che chiama entrambe le funzioni per simulare l'azione completa della pressione di un tasto.

Eseguite il codice seguente, che stamperà un carattere di dollaro (ottenuto tenendo premuto il tasto Maiusc e premendo 4):

```
>>> pyautogui.keyDown('shift'); pyautogui.press('4'); pyautogui.keyUp('shift')
```

Questa riga tiene premuto Maiusc, preme (e rilascia) 4, poi rilascia Maiusc.

Se dovete scrivere una stringa in un campo di testo, la funzione `typewrite()` è più adatta, ma per applicazioni che richiedono comandi a tasto singolo, la funzione `press()` è la strada più semplice.

## Combinazioni di scelta rapida

Una **combinazione di scelta rapida**, o **scorciatoia** da tastiera (o, in inglese, **hotkey**) è una combinazione di tasti che chiama qualche funzione dell'applicazione. La combinazione tipica per copiare è Ctrl-C (in Windows e Linux) o ⌘-C (in OS X). L'utente preme e tiene premuto il tasto Ctrl, poi preme il tasto C, poi rilascia entrambi i tasti. La stessa cosa, con le funzioni `keyDown()` e `keyUp()` di Python, richiede quanto segue.

```
pyautogui.keyDown('ctrl')
pyautogui.keyDown('c')
pyautogui.keyUp('c')
pyautogui.keyUp('ctrl')
```

Un po' complicato, ma la funzione `pyautogui.hotkey()`, che accetta più argomenti con le stringhe dei tasti, li preme in ordine e li rilascia in ordine inverso. Per il caso di Ctrl-C, il codice sarebbe semplicemente:

```
pyautogui.hotkey('ctrl', 'c')
```

Questa funzione è particolarmente utile per le combinazioni più complesse. In Word, la combinazione Ctrl-Alt-Maiusc-S visualizza il pannello degli *Stili*. Anziché otto chiamate di funzione diverse, quattro per `keyDown()` e quattro per `keyUp()`, basta chiamare `hotkey('ctrl', 'alt', 'shift', 's')`.

Con una nuova finestra di file editor nell'angolo superiore sinistro dello schermo, inserite quanto segue nella shell interattiva (in OS X, sostituite 'alt' con 'ctrl'):

```
>>> import pyautogui, time
>>> def commentAfterDelay():
❶    pyautogui.click(100, 100)
❷    pyautogui.typewrite('In IDLE, Alt-3 trasforma una riga in un commento.')
    time.sleep(2)
❸    pyautogui.hotkey('alt', '3')

>>> commentAfterDelay()
```

Questo definisce una funzione `commentAfterDelay()` che, quando chiamata, fa clic sulla finestra del file editor per renderla attiva ❶, scrive *In IDLE, Alt-3 trasforma una riga in un commento.* ❷, entra in pausa per 2 secondi, poi simula la pressione della combinazione di tasti Alt-3 (Ctrl-3 in OS X) ❸. Questa scorciatoia da tastiera aggiunge due caratteri # alla riga corrente, trasformandola in un commento. (È un trucchetto molto utile, quando si scrive del codice in IDLE.)

## Panoramica delle funzioni di PyAutoGUI

Abbiamo passato in rassegna molte funzioni diverse, in questo capitolo. Per comodità di consultazione, ecco qui una rapida panoramica riassuntiva.

- `moveTo(x, y)` Sposta il cursore alle coordinate *x* e *y* date.
- `moveRel(xOffset, yOffset)` Sposta il cursore del mouse relativamente alla posizione corrente.
- `dragTo(x, y)` Sposta il cursore del mouse tenendo premuto il pulsante sinistro.
- `dragRel(xOffset, yOffset)` Sposta il cursore del mouse relativamente alla posizione corrente, tenendo premuto il pulsante sinistro.

- `click(x, y, button)` Simula un clic (del pulsante sinistro per default).
- `rightClick()` Simula un clic del pulsante destro.
- `middleClick()` Simula un clic del pulsante centrale.
- `doubleClick()` Simula un doppio clic del pulsante sinistro.
- `mouseDown(x, y, button)` Simula una pressione del pulsante indicato nella posizione  $x, y$ .
- `mouseUp(x, y, button)` Simula il rilascio del pulsante indicato nella posizione  $x, y$ .
- `scroll(units)` Simula la rotellina di scorrimento. Un argomento positivo fa scorrere verso l'alto, un argomento negativo fa scorrere verso il basso.
- `typewrite(message)` Scrive i caratteri della stringa `message` data.
- `typewrite([key1, key2, key3])` Scrive i caratteri corrispondenti alle stringhe di tasto passate.
- `press(key)` Preme il tasto corrispondente alla stringa passata.
- `keyDown(key)` Simula la pressione del tasto corrispondente alla stringa passata.
- `keyUp(key)` Simula il rilascio del tasto.
- `hotkey([key1, key2, key3])` Simula la pressione dei tasti indicate, nell'ordine indicato, e poi il loro rilascio in ordine inverso.
- `screenshot()` Restituisce una cattura di schermata come oggetto `Image`. (Vedete il [Capitolo 17](#) per maggiori informazioni sugli oggetti `Image`.)

## Progetto: compilatore automatico di moduli

Fra tutte le attività noiose, la compilazione di moduli è la più antipatica. È giusto che sia l'obiettivo di un progetto per l'ultimo capitolo del libro. Supponiamo che abbiate una gran quantità di dati raccolti in un foglio di calcolo, e che vi tocchi il compito di ricopiarli nell'interfaccia a modulo di qualche altra applicazione – e che non ci sia in giro qualche stagista a cui rifilare l'incombenza. Qualche applicazione ha una funzione di importazione che permette di caricare le informazioni da un foglio di calcolo, ma in qualche caso sembra proprio non ci sia altra strada che fare clic e battere tasti per ore senza un briciole di creatività. Ma siete arrivati fino a questo punto del libro e sapete che ovviamente esiste un'altra strada.

Il modulo per questo progetto è un modulo di Google Docs che potete trovare all'indirizzo <http://nostarch.com/automatestuff> e che si presenta come la [Figura 18.4](#).

The screenshot shows a Google Form titled "Generic Form". The form is described as being for the GUI automation project from Chapter 18 of "Automate the Boring Stuff with Python". It contains the following fields:

- A required text input field labeled "Name \*".
- A text input field labeled "Greatest Fear(s)".
- A dropdown menu labeled "What is the source of your wizard powers? \*".
- A statement "Robocop was the greatest action movie of the 1980s" followed by a scale from 1 to 5.
- A rating scale with radio buttons ranging from "Strongly Disagree" to "Strongly Agree".
- A text area labeled "Additional Comments".

At the bottom, there is a "Invia" button, a note about not sending passwords via email, and links for Google Forms and Google's terms of service.

**Figura 18.4** - Il modulo utilizzato per questo progetto.

Ad alto livello, il vostro programma dovrà:

- fare clic sul primo campo di testo del modulo;
- spostarsi nel modulo, scrivendo le informazioni in ciascun campo;
- fare clic sul pulsante *Invia*;
- ripetere il procedimento con il gruppo di dati successivo.

Questo significa che il codice dovrà:

- chiamare `pyautogui.click()` per fare clic sul modulo e sul pulsante *Submit*;
- chiamare `pyautogui.typewrite()` per inserire il testo nei campi;
- gestire l'eccezione `KeyboardInterrupt` in modo che l'utente possa premere `Ctrl-C` per uscire.

Aprite una nuova finestra di file editor e salvate il file con il nome *formFiller.py*.

## Passo 1: stabilire i passi necessari

Prima di scrivere il codice, dove stabilire esattamente i tasti da premere e i clic del mouse per compilare una volta il modulo. Lo script *mouseNow* a [pagina 434](#) può aiutarvi a stabilire le specifiche coordinate del mouse. Dovete sapere le coordinate solo del primo campo di testo. Dopo aver fatto clic sul primo campo, potete semplicemente premere Tab per spostare il focus sul campo successivo. Questo vi risparmierà di dover stabilire le coordinate *x* e *y* per fare clic su ogni campo. Questi sono i passi necessari per inserire i dati nel modulo.

1. Fate clic sul campo *Name*. (Usate *mouseNow.py* per stabilire le coordinate dopo aver ingrandito a tutto schermo la finestra del browser. In OS X, dovete fare clic due volte: una per attivare il browser e una per fare clic sul campo *Name*.)
2. Scrivete un nome e poi premete Tab.
3. Scrivete la cosa più temuta e poi premete Tab.
4. Premete il tasto freccia in giù il giusto numero di volte per selezionare la sorgente di potenza magica: una volta per *wand* (bacchetta), due per *amulet*, tre per *crystal ball*, quattro per *money*. Poi premete Tab. (Notate che, in OS X, dovete premere il tasto freccia in giù una volta di più per ciascuna opzione. Per alcuni browser, dovete premere anche il tasto Invio.)
5. Premete il tasto freccia destra per rispondere alla domanda su Robocop. Premete una volta per 2, due per 3, tre volte per 4 e quattro per 5; oppure premete semplicemente la barra spaziatrice per selezionare *I* (opzione evidenziata per default). Poi premete Tab.
6. Scrivete eventuali commenti ulteriori, quindi premete Tab.
7. Premete il tasto Invio per “fare clic” sul pulsante *Invia*.
8. Dopo aver inviato il modulo, il browser vi porterà a una pagina in cui dovete fare clic su un link per tornare alla pagina del modulo.

Notate che, se eseguite nuovamente questo programma in seguito, dovete aggiornare le coordinate per i clic del mouse, perché la finestra del browser potrebbe avere cambiato posizione.

Per aggirare il problema, assicuratevi sempre che la finestra del browser sia massimizzata a tutto schermo prima di determinare le coordinate del primo campo del modulo. Inoltre, browser diversi sotto sistemi operativi diversi possono funzionare in modo leggermente diverso, perciò controllate bene quali combinazioni di tasti funzionano per il vostro computer, prima di eseguire il programma.

## Passo 2: impostare le coordinate

Caricate il modulo di esempio che avete scaricato ([Figura 18.4](#)) in un browser e massimizzate a tutto schermo la finestra del browser. Aprite una nuova finestra di Terminale o la finestra della riga di comando per eseguire lo script *mouseNow.py*, poi portate il mouse sul campo *Name* per determinarne le coordinate *x* e *y*. Questi numeri verranno assegnati, nel vostro programma, alla variabile *nameField*. Inoltre, stabilite le coordinate *x* e *y* e il valore tupla RGB del pulsante blu *Invia*. Questi valori verranno assegnati alle variabili *submitButton* e *submitButtonColor*, rispettivamente.

Poi, inserite dei dati fintizi nel modulo e fate clic su *Invia*. Dovete vedere come si presenta la pagina successiva, in modo da poter usare *mouseNow.py* per stabilire le coordinate del link *Submit another response* sulla nuova pagina. Scrivete il codice seguente, facendo attenzione a sostituire tutti i valori che qui compaiono in corsivo con le coordinate ottenute dai vostri test:

```
# formFiller.py – Compila automaticamente il modulo.
```

```
import pyautogui, time
```

```
# Qui usate le coordinate giuste per il vostro computer.
```

```
nameField = (648, 319)
```

```
submitButton = (651, 817)
```

```
submitButtonColor = (75, 141, 249)
```

```
submitAnotherLink = (760, 224)
```

```
# DA FARE: Dare all’utente la possibilità di uscire dallo script.
```

```
# DA FARE: Attendere il caricamento della pagina del modulo.
```

```
# DA FARE: Compilare il campo Name.
```

```
# DA FARE: Compilare il campo Greatest Fear(s).
```

```
# DA FARE: Compilare il campo Source of Wizard Powers.
```

```
# DA FARE: Compilare il campo RoboCop.
```

```
# DA FARE: Compilare il campo Additional Comments.
```

```
# DA FARE: Fare clic su Submit.
```

```
# DA FARE: Attendere il caricamento della pagina del modulo.
```

```
# DA FARE: Fare clic sul link Submit another response.
```

Ora vi servono i dati che volete effettivamente inserire in questo modulo. Nel mondo reale, i dati potrebbero arrivare da un foglio di calcolo, da un file di puro testo o da un sito web, e vi servirà del codice ulteriore per caricare i dati. Per questo progetto, semplicemente inseriremo tutti i dati in modo rigido in una variabile. Aggiungete quanto segue al vostro programma:

```
#! python3
# formFiller.py – Compila automaticamente il modulo.

--righe omesse--

formData = [ {'name': 'Alice', 'fear': 'eavesdroppers', 'source': 'wand',
    'robocop': 4, 'comments': 'Tell Bob I said hi.'},
    {'name': 'Bob', 'fear': 'bees', 'source': 'amulet', 'robocop': 4,
    'comments': 'n/a'},
    {'name': 'Carol', 'fear': 'puppets', 'source': 'crystal ball',
    'robocop': 1, 'comments': 'Please take the puppets out of the
    break room.'},
    {'name': 'Alex Murphy', 'fear': 'ED-209', 'source': 'money',
    'robocop': 5, 'comments': 'Protect the innocent. Serve the public
    trust. Uphold the law.'},
    ]
```

```
--righe omesse--
```

La lista `formData` contiene quattro dizionari per quattro nomi diversi. Ciascun dizionario ha i nomi dei campi di testo come chiavi e le risposte come valori. L'ultimo elemento di inizializzazione è impostare la variabile `PAUSE` di PyAutoGUI per un'attesa di mezzo secondo dopo ogni chiamata di funzione. Aggiungete quanto segue al vostro programma dopo l'enunciato di assegnazione `formData`:

```
pyautogui.PAUSE = 0.5
```

## Passo 3: iniziare a scrivere i dati

Un ciclo `for` itererà su ciascuno dei dizionari nella lista `formData`, passando i valori del dizionario alle funzioni di PyAutoGUI che scriveranno virtualmente nei campi di testo. Aggiungete quanto segue al vostro programma:

```
#! python3
# formFiller.py - Compila automaticamente il modulo.

--righe omesse--

for person in formData:
    # Dà all'utente la possibilità di uscire.
    print('">>>> PAUSA DI 5 SECONDI PER CONSENTIRE ALL'UTENTE DI PREMERE CTRL-C <<<')
    time.sleep(5)
    # Attende il caricamento della pagina del modulo.
    ①   while not pyautogui.pixelMatchesColor(submitButton[0], submitButton[1],
    ②       submitButtonColor):
        time.sleep(0.5)
--righe omesse--
```

Come piccola precauzione, lo script ha una pausa di cinque secondi ①, che dà all'utente la possibilità di premere Ctrl-C (o di spostare il cursore nell'angolo superiore sinistro dello schermo per sollevare l'eccezione `FailSafeException`) e chiudere il programma, nel caso stia combinando qualcosa che non va. Poi il programma aspetta finché non è visibile il colore del pulsante *Invia* ②, consentono al programma di sapere che la pagina del modulo è stata scaricata. Ricordate che avete stabilito le informazioni su coordinate e colore nel passo 2 e le avete memorizzate nelle variabili `submitButton` e `submitButtonColor`. Per usare `pixelMatchesColor()`, passate le coordinate `submitButton[0]` e `submitButton[1]`, e il colore `submitButtonColor`.

Dopo il codice che aspetta finché non è visibile il colore del pulsante *Invia*, aggiungete quanto segue:

```
#! python3
# formFiller.py - Compila automaticamente il modulo.
```

--righe omesse--

```
❶ print('Inserisco le info di %s ...' % (person['name']))
❷ pyautogui.click(nameField[0], nameField[1])

❸ # Compila il campo Name.
❹ pyautogui.typewrite(person['name'] + '\t')

❺ # Compila il campo Greatest Fear(s).
❻ pyautogui.typewrite(person['fear'] + '\t')
```

--righe omesse--

Aggiungiamo ogni tanto una chiamata a `print()`, per visualizzare lo stato del programma nella finestra di Terminale per consentire all’utente di sapere che cosa succede ❶.

Poiché il programma sa che il modulo è stato caricato, è il momento di chiamare `click()` per fare clic sul campo *Name* ❷ e `typewrite()` per inserire la stringa in `person['name']` ❸. Alla fine della stringa passata a `typewrite()` viene aggiunto il carattere '\t' per simulare la pressione di Tab, che sposta il focus sul campo successivo. *Greatest Fear(s)*. Un’altra chiamata a `typewrite()` scriverà la stringa presente in `person['fear']` in questo campo e poi premerà il tasto di tabulazione per passare al campo successivo del modulo ❹.

## Passo 4: gestire gli elenchi di opzioni e i pulsanti di comando

Il menu a discesa per la domanda relativa ai “poteri da mago” e i pulsanti di controllo per il campo *RoboCop* sono più complicati da gestire rispetto ai campi di testo. Per fare clic su queste opzioni con il mouse, dovreste stabilire le coordinate *x* e *y* di ciascuna possibile opzione. È più facile usare i tasti freccia per effettuare la selezione. Aggiungete quanto segue al vostro programma:

```
#! python3
# formFiller.py - Compila automaticamente il modulo.
```

--righe omesse--

```
❶ # Compila il campo Source of Wizard Powers.
❷ if person['source'] == 'wand':
    pyautogui.typewrite(['down', '\t'])
❸ elif person['source'] == 'amulet':
    pyautogui.typewrite(['down', 'down', '\t'])
❹ elif person['source'] == 'crystal ball':
    pyautogui.typewrite(['down', 'down', 'down', '\t'])
elif person['source'] == 'money':
    pyautogui.typewrite(['down', 'down', 'down', 'down', '\t'])

# Compila il campo RoboCop.
if person['robocop'] == 1:
    pyautogui.typewrite([' ', '\t'])
elif person['robocop'] == 2:
    pyautogui.typewrite(['right', '\t'])
elif person['robocop'] == 3:
    pyautogui.typewrite(['right', 'right', '\t'])
elif person['robocop'] == 4:
    pyautogui.typewrite(['right', 'right', 'right', '\t'])
elif person['robocop'] == 5:
    pyautogui.typewrite(['right', 'right', 'right', 'right', '\t'])
```

--righe omesse--

Una volta portato il focus sul menu a discesa (ricordate che avete scritto il codice per simulare la pressione del tasto Tab dopo la compilazione del campo Greatest Fear(s)), la pressione del tasto frecci giù vi farà spostare sull'elemento successivo nell'elenco delle scelte. A seconda del valore di person['source'], il programma deve inviare una serie di pressioni del tasto freccia giù prima di una tabulazione per andare al campo successivo. Se il valore della chiave 'source' nel dizionario dell'utente è 'wand' ❶, simuliamo la pressione del tasto freccia giù una volta (per selezionare *Wand*) e poi la pressione di Tab ❷. Se il valore della chiave 'source' è 'amulet', simuliamo la pressione del tasto freccia giù due volte e poi la pressione di Tab, e così via per le altre risposte possibili.

I pulsanti di controllo per la domanda su RoboCop si possono selezionare con i tasti freccia destra o, se volete selezionare la prima possibilità ❸, semplicemente premendo la barra spaziatrice ❹.

## Passo 5: inviare il modulo e attendere

Potete compilare il campo *Additional Comments* con la funzione typewrite() passando come argomento person['comments']. Potete scrivere un ulteriore '\t' per spostare il focus sul campo successivo o sul pulsante *Invia*. Quando il focus è sul pulsante *Invia*, la chiamata pyautogui.press('enter') simulerà la pressione del tasto Invio e quindi l'invio del modulo. Poi, il programma aspetterà per cinque secondi il caricamento della pagina successiva.

Quando si sarà caricata la nuova pagina, avrà un link *Submit another response* che indirizzerà il browser a una nuova pagina con un modulo in bianco. Avete memorizzato le coordinate di questo link

come tupla in submitAnotherLink nel passo 2, perciò passate queste coordinate a pyautogui.click() per fare clic su questo link.

Quando il nuovo modulo è pronto per la compilazione, il ciclo for esterno dello script può passare all'iterazione successiva e inserire le informazioni relative alla persona successiva. Completate il programma aggiungendo il codice seguente:

```
#! python3
# formFiller.py - Compila automaticamente il modulo.

--righe omesse--

# Compila il campo Additional Comments.
pyautogui.typewrite(person['comments'] + '\t')

# Fa clic su Invia.
pyautogui.press('enter')

# Attende il caricamento della pagina del modulo.
print('Ho fatto clic su Invia.')
time.sleep(5)

# Fa clic sul link Submit another response.
pyautogui.click(submitAnotherLink[0], submitAnotherLink[1])
```

Quando il ciclo for è terminato, il programma avrà inserito le informazioni per tutte le persone. In questo esempio, le persone di cui inserire i dati sono solo quattro, ma se fossero 4000, scrivere un programma che faccia tutte queste cose vi farebbe risparmiare una gran quantità di tempo e di movimenti delle dita.

## Riepilogo

L'**automazione GUI** con il modulo pyautogui consente di **interagire** con le applicazioni presenti sul computer **controllando il mouse e la tastiera**. Questa impostazione è abbastanza flessibile da fare qualsiasi cosa possa fare un essere umano, ma l'aspetto negativo è che questi programmi non “sanno” dove fanno clic o scrivono. Quando si scrivono programmi di automazione GUI, bisogna premunirsi perché falliscano rapidamente, se ricevono istruzioni sbagliate. Un crash è irritante, ma è molto meglio che avere un programma che continua a fare errori.

Potete spostare il cursore del mouse per lo schermo e simulare clic del mouse, pressioni di tasti e scorciatoie da tastiera on il modulo pyautogui, che può anche controllare i colori sullo schermo, il che darà ai vostri programmi di automazione un’idea abbastanza buona dei contenuti dello schermo da sapere se è finito fuori dal seminato. Potete addirittura dare a PyAutoGUI una schermata e fare in modo che stabilisca le coordinate dell’area in cui volete fare clic.

Potete combinare tutte queste funzionalità di PyAutoGUI per automatizzare qualsiasi compito noiosamente ripetitivo. In effetti, può avere un effetto ipnotico osservare il cursore del mouse che si sposta da solo e vedere il testo comparire sullo schermo automaticamente. Perché non passare il tempo risparmiato standosene seduti a vedere come il vostro programma faccia tutto il lavoro per voi? Si prova una certa soddisfazione nel vedere come un po’ di abilità abbia fatto risparmiare un bel

po' di lavoro noioso.

## Domande di ripasso

1. Come si può abilitare la funzionalità “fail safe” di PyAutoGUI per fermare un programma?
2. Quale funzione restituisce la `resolution()` corrente?
3. Quale funzione restituisce le coordinate della posizione corrente del mouse?
4. Qual è la differenza fra `pyautogui.moveTo()` e `pyautogui.moveRel()`?
5. Quali funzioni si possono usare per trascinare il mouse?
6. Quale chiamata di funzione scriverà i caratteri di "Hello world!"?
7. Come si può simulare la pressione di tasti speciali come il tasto freccia sinistra?
8. Come si possono salvare i contenuti correnti dello schermo in un file di immagine con il nome `screenshot.png`?
9. Che codice imposterebbe una pausa di due secondi dopo ogni chiamata a una funzione di PyAutoGUI?

## Un po' di pratica

Per esercitarvi, scrivete dei programmi che svolgano le attività seguenti.

### Far finta di essere occupati

Molti programmi di instant messaging stabiliscono se non state facendo nulla, o se siete lontani dal computer, identificando l'assenza di spostamenti del mouse in un certo arco di tempo, per esempio dieci minuti. Magari vorreste allontanarvi per un po' dalla scrivania, ma non volete che gli altri vedano lo stato del programma di messaggistica passare in modalità “idle”. Scrivete uno script che sposti leggermente il cursore del mouse ogni dieci secondi. Lo spostamento dovrebbe essere abbastanza piccolo da non essere d'intralcio se dovete usare il computer mentre lo script è in esecuzione.

### Bot per la messaggistica istantanea

Google Talk, Skype, Yahoo Messenger, AIM e altre applicazioni di messaggistica istantanea usano spesso protocolli proprietari che rendono difficile la scrittura di moduli Python in grado di interagire con questi programmi, ma nemmeno questi protocolli proprietari possono impedirvi di scrivere uno strumento di automazione GUI.

L'applicazione Google Talk ha una barra di ricerca che consente di inserire il nome utente di qualcuno che fa parte della lista degli amici e di aprire una finestra di messaggio quando si preme Invio. Il focus della tastiera passa automaticamente alla nuova finestra. Altre applicazioni di messaggistica hanno modi simili per aprire una nuova finestra di messaggio. Scrivete un programma che invii automaticamente un messaggio di notifica a un gruppo selezionato di persone scelte nell'elenco dei vostri amici. Il programma deve poter gestire casi eccezionali, come amici che sono offline, la finestra di chat che compare a coordinate diverse sullo schermo, o le finestre di conferma che interrompono l'inserimento di messaggi. Il programma dovrà prendere delle catture di schermo per orientare l'interazione con la GUI e adottare dei modi per stabilire quando le sue pressioni di tasti virtuali non vengono inviate.

## NOTA

Potreste creare qualche account di prova fasullo, in modo da non inondare accidentalmente di spam i vostri amici reali mentre scrivete il programma.

### Bot tutorial di gioco

Esiste un eccellente tutorial intitolato “How to Build a Python Bot That Can Play Web Games”, che potete trovare all’indirizzo <http://nostarch.com/automatestuff>. Questo tutorial spiega come creare in Python un programma di automazione GUI che gioca a Sushi Go Round, un gioco in Flash. Si gioca facendo clic sui pulsanti degli ingredienti giusti per soddisfare gli ordini di sushi dei clienti. Quanto più rapidamente si evadono gli ordini senza commettere errori, tanti più punti si guadagnano. È un compito perfetto per un programma di automazione GUI, e anche un modo per barare e ottenere un punteggio elevato. Il tutorial affronta molti argomenti analoghi a quelli di questo capitolo, ma esamina anche le funzionalità di base per il riconoscimento delle immagini di PyAutoGUI.

# Installare moduli di terze parti

Oltre alla **libreria standard** di moduli forniti con Python, altri sviluppatori hanno scritto propri **moduli** per estendere ulteriormente le capacità di Python. Il metodo principale per **installare** moduli di terze parti è usare lo **strumento pip** di Python.

Questo strumento scarica e installa in modo sicuro i moduli Python sul vostro computer da <https://pypi.python.org/>, il sito web della pYthon Software Foundation. PyPI, Python Packe Index, è una sorta di app store libero per moduli Python.

## Lo strumento pip

Il file eseguibile dello strumento pip si chiama *pip* in Windows e *pip3* in OS X e Linux. In Windows lo si trova nella directory *C:\Python35-32\Scripts\pip.exe*; in OS X è in */Library/Frameworks/Python.framework/Versions/3.5/bin/pip3*. In Linux, è in */usr/bin/pip3*.

Mentre pip viene automaticamente installato nelle versioni di Python 3.x sotto Windows e OS X, bisogna installarlo separatamente in Linux. Per installare pip3 sotto Ubuntu o Debian Linux, aprire una nuova finestra di Terminale, inserire `sudo apt-get install python3-pip`. Per installare pip3 sotto Fedora Linux, inserire `sudo yum install python3-pip` in una finestra di Terminale. Dovrete inserire la password di amministratore del computer per installare questo software.

## Installare moduli di terze parti

Lo strumento pip deve essere eseguito dalla riga di comando: gli si passa il comando `install` seguito dal nome del modulo che si vuole installare. Per esempio, in Windows si inserisce `pip install NomeModulo`, dove *NomeModulo* è il nome del modulo da installare. In OS X e in Linux, bisogna eseguire pip3 con il prefisso `sudo` per ottenere i privilegi di amministratore necessari per installare il modulo. Dovete scrivere `sudo pip3 install NomeModulo`.

Se avete già installato il modulo, ma volete aggiornarlo alla versione più recente disponibile in PyPI, eseguite `pip install -U NomeModulo` (o `sudo pip3 install -U NomeModulo` sotto OS X e Linux).

Dopo aver installato il modulo, potete verificare che sia stato installato correttamente eseguendo `import NomeModulo` nella shell interattiva. Se non vengono visualizzati messaggi d'errore, potete dare per scontato che il modulo sia stato installato correttamente.

Potete installare tutti i moduli utilizzati in questo libro eseguendo la lista di comandi qui di seguito. (Ricordate di sostituire `pip` con `pip3` se usate OS X o Linux.)

- `pip install send2trash`
- `pip install requests`
- `pip install beautifulsoup4`
- `pip install selenium`
- `pip install openpyxl==2.1.4`
- `pip install PyPDF2`
- `pip install python-docx` (**installate** `python-docx`, non `docx`)
- `pip install imapclient`
- `pip install pyzmail`
- `pip install twilio`
- `pip install pillow`
- `pip install pyobjc-core` (**solo OS X**)
- `pip install pyobjc` (**solo OS X**)
- `pip install python3-xlib` (**solo Linux**)
- `pip install pyautogui`

## NOTA

Per gli utenti di OS X: il modulo `pyobjc` può richiedere anche 20 minuti o più per installarsi, perciò non vi preoccupate se richiede un po' di tempo. Dovete anche installare prima il modulo `pyobjc-core`, che ridurrà il tempo complessivo di installazione.

# Eseguire i programmi

Se avete un programma aperto nel file editor di IDLE, per eseguirlo basta premere **F5** oppure selezionare da menu **Run > Run Module**. Questo è un modo semplice per eseguire i programmi mentre li si scrive, ma aprire IDLE per eseguire programmi finiti può essere seccante. Esistono **modi più comodi** per eseguire gli script Python.

## La riga shebang

La prima riga di tutti programmi Python deve essere una **riga shebang**, che dice al computer che volete che il programma sia eseguito da Python. La riga shebang inizia con `#!`, ma il resto dipende dal sistema operativo.

- In Windows, la riga shebang è `#! Python3`.
- In OS X, la riga shebang è `#! /usr/bin/env python3`.
- In Windows, la riga shebang è `#! /usr/bin/python3`.

Potete eseguire gli script Python da IDLE senza questa riga, ma è necessaria invece per eseguirli dalla riga di comando.

## Eseguire programmi Python in Windows

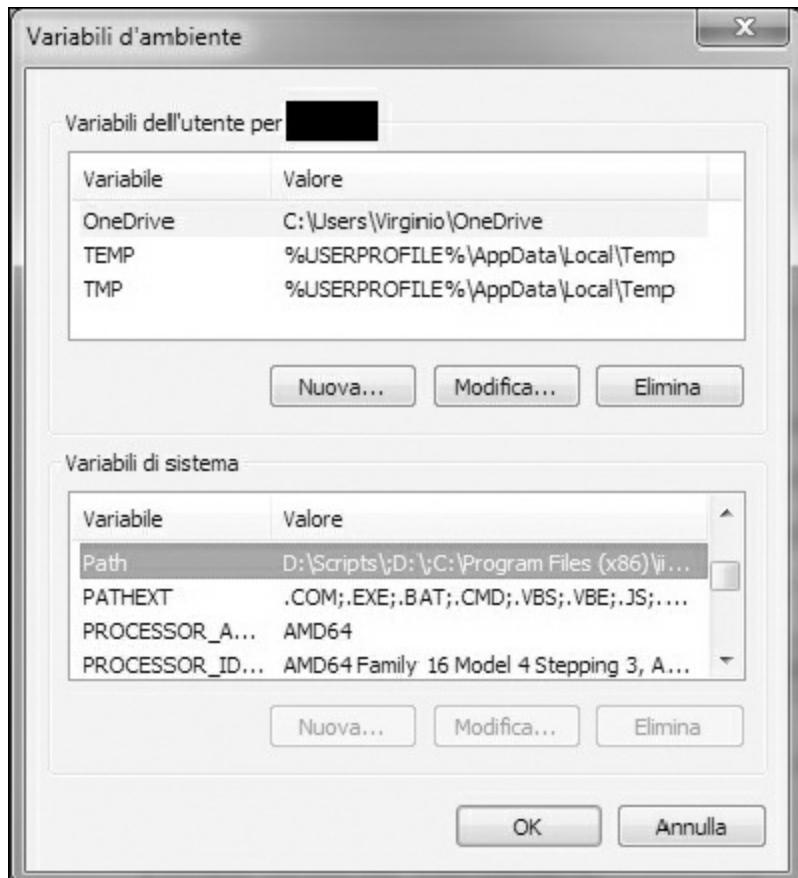
In Windows, l'interprete Python 3.x si trova in `C:\Python35-32\python.exe`. In alternativa, il comodo programma `py.exe` leggerà la riga shebang all'inizio del codice sorgente del file `.py` ed eseguirà la versione opportuna di Python per quello script. Il programma `py.exe` farà in modo che il programma Python venga eseguito dalla versione corretta di Python, se sul computer sono installate più versioni. Per rendere più comodo eseguire il vostro programma Python, create un file batch con estensione `.bat` per eseguirlo con `py.exe`. Per creare il file batch, preparate un nuovo file di testo che contiene una

sola riga, come il seguente:

```
@py.exe C:\percorso\alvostro\pythonScript.py %*
```

Sostituite il percorso segnaposto con il percorso assoluto del vostro programma e salvate il file con estensione *.bat* (per esempio, *python.Script.bat*). Questo file batch vi risparmierà di dover scrivere il percorso assoluto del programma Python ogni volta che volete eseguirlo. Vi consiglio di collocare tutti i vostri file batch e *.py* in un'unica cartella, per esempio *C:\MyPythonScripts* o *C:\Users\VostroNome\PythonScripts*.

La cartella *C:\MyPythonScripts* va aggiunta al percorso di sistema in Windows in modo da poter eseguire i file batch che contiene dalla finestra di dialogo *Esegui*. Per fare questo, bisogna modificare la variabile d'ambiente *PATH*. Fate clic sul pulsante *Start* e scrivete *Modifica variabili di ambiente per l'account*. Questa opzione si dovrebbe completare automaticamente prima che finiate di scriverla. La finestra delle *Variabili di ambiente* che compare si presenta come la [Figura B.1](#).



**Figura B.1** - La finestra Variabili di ambiente in Windows.

Dall'elenco delle variabili di sistema, selezionate la variabile *Path* e fate clic su *Modifica*. Nel campo di testo *Valore*, andate in fondo, accodate un punto e virgola, scrivete *C:\MyPythonScripts*, poi fate clic su *OK*. Ora potete eseguire qualsiasi script Python che si trova nella cartella *C:\MyPythonScripts* semplicemente premendo Win-R e scrivendo il nome dello script. L'esecuzione di *pythonScript*, per esempio, eseguirà *pythonScript.bat*, che a sua volta vi risparmierà di dover inserire l'intero comando *py.exe C:\MyPythonScripts\pythonScript.py* dalla finestra di dialogo *Esegui*.

## Eseguire programmi Python in OS X e Linux

In OS X, selezionando **Applicazioni > Utility > Terminale** si aprirà una finestra di Terminale, che offre la possibilità di inserire comandi utilizzando solo i caratteri, anziché utilizzare un'interfaccia grafica. Per aprire la finestra di Terminale in Ubuntu Linux, premete il tasto Win (o Super) per aprire Dash e scrivete Terminale.

La finestra *Terminale* si attiverà nella cartella home del vostro account utente. Se il mio nome utente è *asweigart*, la cartella home sarà */Users/asweigart* in OS X e */home/asweigart* in Linux. Il carattere tilde (~) è una scorciatoia per la cartella home, perciò potete scrivere `cd ~` per passare alla vostra cartella home. Potete usare il comando `cd` anche per passare dalla directory di lavoro corrente a qualsiasi altra directory. In OS X come in Linux, il comando `pwd` stamperà la directory di lavoro corrente.

Per eseguire i programmi Python, salvate il vostro file *.py* nella cartella home. Poi modificate le autorizzazioni dei file *.py* per renderli eseguibili con il comando `chmod +x pythonScript.py`. Le autorizzazioni per i file sono un tema che esce dall'ambito di questo libro, ma dovete eseguire questo comando sul vostro file Python se volete eseguire il programma dalla finestra di Terminale. Una volta fatto, potrete eseguire il vostro script ogni volta che vorrete, aprendo una finestra di terminale e inserendo `./pythonScript.py`. La riga shebang all'inizio dello script dirà al sistema operativo dove trovare l'interprete Python.

## Eseguire programmi Python con le asserzioni disattivate

Potete disattivare gli enunciati `assert` nei programmi Python, per ottenere un piccolo miglioramento delle prestazioni. Eseguendo Python da terminale, includete il commutatore `-O` dopo `python` o `python3` e prima del nome del file *.py*. Questo manderà in esecuzione una versione ottimizzata del programma, che salta la verifica delle asserzioni.

# Risposte alle domande di ripasso

Questa appendice contiene le **risposte alle domande di ripasso** che si trovano alla fine di ciascun capitolo. Vi consiglio di dedicare del tempo a rispondere: programmare non significa solo memorizzare la sintassi e un elenco di nomi di funzione. Come quando si studia una lingua straniera, più ci si esercita e più se ne ricava. Esistono molti **siti web** che offrono **problemi di programmazione** per esercitarsi: potete trovarne un elenco all'indirizzo <http://nostarch.com/automatestuff/>.

## Capitolo 1

1. Gli operatori sono +, -, \* e /. I valori sono 'hello', -88.8 e 5.
2. La stringa è 'spam', la variabile spam. Le stringhe devono sempre iniziare e finire con apici.
3. I tre tipi di dati introdotti in questo capitolo sono gli interi, i numeri in virgola mobile e le stringhe.
4. Un'espressione è una combinazione di valori e operatori. Tutte le espressioni vengono valutate (cioè si riducono) a un singolo valore.
5. Un'espressione viene valutata a un singolo valore, un enunciato no.
6. La variabile bacon è impostata a 20. L'espressione bacon + 1 non assegna un nuovo valore a bacon (sarebbe stato necessario un enunciato di assegnazione: bacon = bacon + 1).
7. Entrambe le espressioni hanno come valore la stringa 'spamspamspam'.
8. I nomi di variabili non possono iniziare con un numero.
9. Le funzioni int(), float() e str() daranno come valore le versioni intera, in virgola mobile e stringa del valore che viene loro passato.
10. L'espressione provoca un errore perché 99 è un intero, mentre solo stringhe si possono

concatenare ad altre stringhe con l'operatore +. La formulazione corretta sarebbe 'Ho mangiato ' + str(99) + ' burritos.'

## Capitolo 2

1. True e False, con le iniziali T e F maiuscole, il resto della parola in minuscole.

2. and, or e not

3. True and True è True.

True and False è False.

False and True è False.

False and False è False.

True or True è True.

True or False è True.

False or True è True.

False or False è False.

not True è False.

not False è True.

4. False

False

True

False

False

True

5. ==, !=, <, >, <= e >=.

6. == è l'operatore di uguaglianza, che confronta due valori e dà come risultato un Booleano, mentre = è l'operatore di assegnazione, che memorizza un valore in una variabile.

7. Una condizione è un'espressione usata in un enunciato di controllo del flusso che ha come valore un valore Booleano.

8. I tre blocchi sono tutto ciò che sta all'interno dell'enunciato if e le righe print('bacon') e print('ham').

```
print('eggs')
if spam > 5:
    print('bacon')
else:
    print('ham')
print('spam')
```

9. Il codice:

```
if spam == 1:
    print('Hello')
elif spam == 2:
    print('Howdy')
else:
    print('Greetings!')
```

10. Per uscire da un programma che è incappato in un ciclo infinito bisogna premere Ctr-C.

11. L'enunciato `break` trasferirà l'esecuzione al di fuori di un ciclo, subito dopo il ciclo stesso.  
L'enunciato `continue` trasferirà l'esecuzione all'inizio del ciclo.
12. Fanno tutte la stessa cosa. La chiamata `range(10)` va da 0 fino a 10 (escluso), `range(0, 10)` dice esplicitamente al ciclo di partire da 0 e `range(0, 10, 1)` dice esplicitamente al ciclo di incrementare la variabile di 1 a ogni iterazione.

13. Il codice:

```
for i in range(1, 11):  
    print(i)
```

e:

```
i = 1  
while i <= 10:  
    print(i)  
    i = i + 1
```

14. Questa funzione può essere chiamata con `spam.bacon()`.

## Capitolo 3

1. Le funzioni riducono la necessità di duplicare il codice. Così i programmi sono più brevi, più facili da leggere e da aggiornare.
2. Il codice in una funzione viene eseguito quando la funzione viene chiamata, non quando la funzione viene definita.
3. L'enunciato `def` definisce (cioè crea) una funzione.
4. Una funzione è costituita dall'enunciato `def` e dal codice nella sua clausola `def`. Una chiamata di funzione è ciò che trasferisce l'esecuzione del programma all'interno della funzione e la chiamata di funzione ha come valore il valore restituito dalla funzione.
5. Esiste un solo ambito globale; un ambito locale viene creato ogni volta che viene chiamata una funzione.
6. Quando una funzione ritorna, l'ambito locale viene distrutto e tutte le variabili al suo interno sono dimenticate.
7. Un valore di ritorno è il valore a cui viene valutata una chiamata di funzione. Come ogni valore, un valore restituito può essere utilizzato come parte di un'espressione.
8. Se non esiste enunciato di ritorno per una funzione, il valore che restituisce è `None`.
9. Un enunciato `global` costringe una variabile in una funzione a fare riferimento alla variabile globale.
10. Il tipo di dati di `None` è `NoneType`.
11. Quell'enunciato `import` importa un modulo chiamato `arealyourpetsnamederic`. (Nel caso vi fosse venuto il dubbio, non è un modulo Python realmente esistente.)
12. Questa funzione può essere chiamata con `spam.bacon()`.
13. Racchiudete in una clausola `try` la riga di codice che può provocare un errore.
14. Il codice che potrebbe provocare un errore va nella clausola `try`. Il codice che deve essere

eseguito se si verifica un errore va nella clausola except.

## Capitolo 4

1. Il valore lista vuota, che è un valore lista che non contiene alcun elemento. È un po' come il valore stringa vuota “”.
2. spam[2] = 'hello' (notate che il terzo valore in una lista si trova all'indice 2 perché il primo indice è 0).
3. 'd' (notate che '3' \* 2 è la stringa '33', che viene passata a int() prima di essere divisa per 11, dando alla fine il valore 3. Le espressioni si possono usare ovunque si usano i valori).
4. 'd' (i valori negativi contano dal fondo).
5. ['a', 'b']
6. 1
7. [3.14, 'cat', 11, 'cat', True, 99]
8. [3.14, 11, 'cat', True]
9. L'operatore per la concatenazione di liste è +, mentre l'operatore per la replica è \*. (È come per le stringhe.)
10. Mentre append() accoda i valori solo alla fine di una lista, insert() può aggiungerli in qualunque punto della lista.
11. L'enunciato del e il metodo di lista remove() sono i due modi per eliminare valori da una lista.
12. Sia le liste che le stringhe possono essere passate a len(), hanno indici e sezioni, possono essere usate nei cicli for, possono essere concatenate o replicate e possono essere usate con gli operatori in e not in.
13. Le liste possono cambiare; vi si possono aggiungere, eliminare o modificare valori. Le tuple sono immutabili; non si possono cambiare in alcun modo. Le tuple inoltre vengono scritte fra parentesi tonde, mentre le liste usano le parentesi quadre.
14. (42,) (la virgola è obbligatoria).
15. Le funzioni tuple() e list(), rispettivamente.
16. Contengono riferimenti a valori lista.
17. La funzione copy.copy() effettua una copia superficiale di una lista, mentre la funzione copy.deepcopy() effettua una copia profonda. Solo copy.deepcopy(), cioè, duplica le eventuali liste contenute all'interno della lista.

## Capitolo 5

1. Due parentesi graffe: {}.
2. {'foo': 42}
3. Gli elementi memorizzati in un dizionario non sono ordinati, mentre gli elementi di una lista sono ordinati.
4. Si ottiene un errore KeyError.
5. Non c'è differenza. L'operatore in verifica se un certo valore è presente come chiave nel dizionario.
6. 'cat' in spam verifica se esiste una chiave 'cat' nel dizionario, mentre 'cat' in spam.values() verifica se

esiste un valore 'cat' per una delle chiavi in spam.

7. `spam.setdefault('color', 'black')`
8. `pprint.pprint()`

## Capitolo 6

1. I caratteri escape rappresentano caratteri in valori stringa che altrimenti sarebbero difficili o impossibili da scrivere nel codice.
2. `\n` è un carattere newline (a capo); `\t` è un carattere di tabulazione.
3. Il carattere escape `\` rappresenta un carattere barra retroversa.
4. La virgoletta singola in Howl's va bene perché la stringa è stata racchiusa fra apici doppi.
5. Le stringhe multiriga consentono di usare gli a capo nelle stringhe senza il carattere escape `\n`.
6. Le espressioni hanno questi valori:
  - 'i'
  - 'Ciao'
  - 'Ciao'
  - 'o mondo!'
7. Le espressioni hanno questi valori:
  - 'CIAO'
  - True
  - 'ciao'
8. Le espressioni hanno questi valori:
  - ['Ricorda,', 'ricorda,', 'il', 'cinque', 'di', 'Novembre.']}
  - 'Ce-ne-può-essere-solo-uno.'
9. I metodi stringa `rjust()`, `ljust()` e `center()`, rispettivamente.
10. I metodi `lstrip()` e `rstrip()` eliminano gli spazi bianchi rispettivamente all'estremità sinistra e destra di una stringa.

## Capitolo 7

1. La funzione `re.compile()` restituisce oggetti Regex.
2. Si usano le stringhe grezze per non dover usare caratteri escape per le barre retroverse.
3. Il metodo `search()` restituisce oggetti Match.
4. Il metodo `group()` restituisce stringhe del testo individuato.
5. Il gruppo 0 è la corrispondenza completa, il gruppo 1 comprende il primo insieme di parentesi, il gruppo 2 il secondo insieme di parentesi.
6. Punti e parentesi possono essere inseriti come caratteri escape con una barra retroversa: `\.`, `\(` e `\)`.
7. Se l'espressione regolare non ha gruppi, viene restituita una lista di stringhe. Se l'espressione contiene gruppi, viene restituita una lista di tuple di stringhe.
8. Il carattere `|` significa una corrispondenza “o, o” fra i due gruppi.
9. Il carattere `?` può significare o “cerca zero o una occorrenza del gruppo precedente” oppure può essere usato per indicare una corrispondenza non avida.

10. Il + corrisponde a “uno o più”, l’asterisco \* a “zero o più”.
11. La notazione {3} indica la ricerca di una corrispondenza esattamente con tre occorrenze del gruppo precedente. {3,5} indica la corrispondenza con fra tre e cinque occorrenze.
12. Le classi di caratteri \d, \w e \s corrispondono a una singola cifra, una singola parola o un singolo carattere spazio, rispettivamente.
13. Le classi di caratteri \D, \W e \S corrispondono a un singolo carattere che non sia una cifra, non sia una parola o non sia un carattere spazio, rispettivamente.
14. Passando re.I o re.IGNORECASE come secondo argomento a re.compile(), la ricerca non farà differenza fra maiuscole e minuscole.
15. Il carattere . normalmente corrisponde a qualsiasi carattere tranne il carattere di a capo (newline). Se si passa re.DOTALL come secondo argomento a re.compile(), il punto corrisponde anche ai caratteri di a capo.
16. L’espressione .\* effettua una ricerca avida, .\*? una ricerca non avida.
17. [0-9a-z] oppure [a-zA-Z].
18. 'X drummers, X pipers, five rings, X hens'
19. L’argomento re.VERBOSE consente di aggiungere spazi bianchi e commenti alla stringa passata a re.compile().
20. re.compile(r'^d{1,3}(\d{3})\*\$') creerà questa espressione regolare, ma altre stringhe possono produrre un’espressione regolare simile.
21. re.compile(r'[A-Z][a-z]\*'sNakamoto')
22. re.compile(r'(Alice|Bob|Carol)\s(mangia|coccoca|lancia)\s(mele|gatti|palline)\.', re.IGNORECASE)

## Capitolo 8

1. I percorsi relativi sono relativi alla directory di lavoro corrente.
2. I percorsi assoluti iniziano con la cartella radice, per esempio / o C:\.
3. La funzione os.getcwd() restituisce la directory di lavoro corrente. La funzione os.chdir() cambia la directory di lavoro corrente.
4. La cartella . è la cartella corrente, .. è la cartella genitrice.
5. C:\bacon\eggs è il nome della directory, spam.txt è il nome base.
6. La stringa 'r' per la modalità di lettura, 'w' per la modalità di scrittura, 'a' per la modalità accodamento.
7. Un file esistente aperto in modalità scrittura viene cancellato e sovrascritto.
8. Il metodo read() restituisce tutti i contenuti del file come un unico valore stringa. Il metodo readlines() restituisce una lista di stringhe, dove ciascuna stringa è una riga dei contenuti del file.
9. Un valore shelf assomiglia a un valore dizionario; possiede chiavi e valori e metodi keys() e values() che funzionano in modo analogo ai metodi di dizionario che hanno lo stesso nome.

## Capitolo 9

1. La funzione shutil.copy() copia un singolo file, mentre shutil.copytree() copia un’intera cartella, con tutti i suoi contenuti.

2. La funzione `shutil.move()` si usa per cambiare nome ai file e anche per spostarli.
3. Le funzioni `send2trash` spostano un file o una cartella al cestino, mentre le funzioni `shutil` cancellano in modo definitivo file e cartelle.
4. La funzione `zipfile.ZipFile()` è equivalente alla funzione `open()`; il primo argomento è il nome del file, il secondo argomento è il modo in cui deve essere aperto il file (lettura, scrittura o accodamento).

## Capitolo 10

1. `assert(spam >= 10, 'The spam variable is less than 10.')`
2. `assert(eggs.lower() != bacon.lower(), 'The eggs and bacon variables are the same!')` O `assert(eggs.upper() != bacon.upper(), 'The eggs and bacon variables are the same!')`
3. `assert(False, 'This assertion always triggers.')`
4. Per poter chiamare `logging.debug()` bisogna avere queste due righe all'inizio del programma:

```
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -
%(levelname)s - %(message)s')
```

5. Per poter inviare messaggi di log a un file che si chiama *programLog.txt* con `logging.debug()`, bisogna avere queste due righe all'inizio del programma:

```
import logging
logging.basicConfig(filename='programLog.txt', level=logging.DEBUG,
format='%(asctime)s - %(levelname)s - %(message)s')
```

6. DEBUG, INFO, WARNING, ERROR, CRITICAL.
7. `logging.disable(logging.CRITICAL)`
8. Si possono disattivare i messaggi di logging senza eliminare le chiamate delle funzioni di logging. Si possono disattivare selettivamente i messaggi di logging di livello inferiore. Si possono creare messaggi di logging. I messaggi di logging includono l'indicazione di data e ora.
9. Il pulsante *Step* fa passare il debugger all'interno di una chiamata di funzione. Il pulsante *Over* eseguirà rapidamente la chiamata di funzione senza procedere a passi nel suo interno. Il pulsante *Out* eseguirà rapidamente il resto del codice finché non esce dalla funzione in cui si trova in quel momento.
10. Dopo aver fatto clic su *Go*, il debugger si fermerà quando raggiunge la fine del programma o una riga con un punto di interruzione.
11. Un punto di interruzione o breakpoint è un'impostazione, su una riga di codice, che fa sì che il debugger si fermi quando l'esecuzione del programma raggiunge quella riga.
12. Per impostare un punto di interruzione in IDLE, si fa clic destro sulla riga e si seleziona *Set Breakpoint* dal menu di scelta rapida.

## Capitolo 11

1. Il modulo `webbrowser` ha un metodo `open()` che lancia un browser web su un URL specifico, e

questo è tutto. Il modulo requests può scaricare file e pagine dal Web. Il modulo BeautifulSoup analizza l'HTML. Infine, il modulo selenium può lanciare e controllare un browser.

2. La funzione `requests.get()` restituisce un oggetto Response, che ha un attributo `text` che contiene i contenuti scaricati, sotto forma di stringa.
3. Il metodo `raise_for_status()` solleva un'eccezione se il download ha problemi e non fa nulla se il download va a buon fine.
4. L'attributo `status_code` dell'oggetto Response contiene il codice di status http.
5. Dopo aver aperto il nuovo file in modalità scrittura 'wb', si usa un ciclo for che itera sul metodo `iter_content()` dell'oggetto Response per scrivere a blocchi sul file. Ecco un esempio:

```
saveFile = open('filename.html', 'wb')
for chunk in res.iter_content(100000):
    saveFile.write(chunk)
```

6. F12 apre gli strumenti per gli sviluppatori in Chrome. Premendo Ctrl-Maiusc-C (in Windows e Linux) o ⌘-opzione-C (in OS X) si aprono gli strumenti per gli sviluppatori in Firefox.
7. Si fa clic destro sull'elemento nella pagina e si seleziona *Ispeziona elemento* dal menu.
8. '#main'
9. '.highlight'
10. 'div div'
11. 'button[value="favorite"]'
12. spam.getText()
13. linkElem.attrs
14. Il modulo selenium si importa con `from selenium import webdriver`.
15. I metodi `find_element_*` restituiscono il primo elemento corrispondente come oggetto WebElement. I metodi `find_elements_*` restituiscono una lista di tutti gli elementi corrispondenti come oggetti WebElement.
16. I metodi `click()` e `send_keys()` simulano clic del mouse e pressioni di tasto alla tastiera, rispettivamente.
17. Chiamando il metodo `submit()` su qualsiasi elemento all'interno di un modulo si invia il modulo.
18. I metodi `forward()`, `back()` e `refresh()` dell'oggetto WebDriver simulano i pulsanti del browser.

## Capitolo 12

1. La funzione `openpyxl.load_workbook()` restituisce un oggetto Workbook.
2. Il metodo `get_sheet_names()` restituisce un oggetto Worksheet.
3. Chiamate `wb.get_sheet_by_name('Foglio1')`.
4. Chiamate `wb.get_active_sheet()`.
5. `sheet['C5'].value` O `sheet.cell(row=5, column=3).value`
6. `sheet['C5'] = 'Hello'` O `sheet.cell(row=5, column=3).value = 'Hello'`
7. `cell.row` e `cell.column`
8. Restituiscono sotto forma di valori interi rispettivamente l'ultima colonna e l'ultima riga del foglio che contengono valori.

9. `openpyxl.cell.column_index_from_string('M')`
10. `openpyxl.cell.get_column_letter(14)`
11. `sheet['A1':'F1']`
12. `wb.save('example.xlsx')`
13. Una formula si imposta come qualsiasi valore. Si imposta l'attributo `value` a una stringa con il testo della formula. Ricordate che le formule iniziano sempre con il segno `=`.
14. Quando chiamate `load_workbook()`, passate `True` per l'argomento per parola chiave `data_only`.
15. `sheet.row_dimensions[5].height = 100`
16. `sheet.column_dimensions['C'].hidden = True`
17. OpenPyXL 2.0.5 non carica riquadri bloccati, titoli per la stampa, immagini o grafici.
18. I riquadri bloccati sono righe e colonne che compaiono sempre sullo schermo. Sono utili per mantenere sempre visibili le intestazioni.
19. `openpyxl.charts.Reference()`, `openpyxl.charts.Series()`, `openpyxl.charts.BarChart()`, `chartObj.append(seriesObj)` e `add_chart()`.

## Capitolo 13

1. Un oggetto `File` restituito da `open()`.
2. Lettura binaria ('rb') per `PdfFileReader()` e scrittura binaria ('wb') per `PdfFileWriter()`.
3. Chiamando `getPage(4)` si ottiene di ritorno un oggetto `Page` per la pagina 5, poiché la pagina 0 è la prima.
4. La variabile `numPages` memorizza un intero che è il numero delle pagine nell'oggetto `PdfFileReader`.
5. Chiamate `decrypt('swordfish')`.
6. I metodi `rotateClockwise()` e `rotateCounterClockwise()`. Come argomento si passa un intero che sono i gradi di cui effettuare la rotazione.
7. `docx.Document('demo.docx')`
8. Un documento contiene più paragrafi. Un paragrafo inizia su una nuova riga e contiene più run. I run sono gruppi contigui di caratteri entro un paragrafo.
9. Usate `doc.paragraphs`.
10. Un oggetto `Run` ha tre variabili (non un paragrafo).
11. `True` rende sempre l'oggetto `Run` in grassetto e `False` lo rende sempre non in grassetto, indipendentemente da quale sia l'impostazione dello stile. Non farà in modo che l'oggetto `Run` usi l'impostazione dello stile per il grassetto.
12. Chiamate la funzione `docx.Document()`.
13. `doc.add_paragraph('Hello there!')`
14. Gli interi 0, 1, 2, 3 e 4.

## Capitolo 14

1. In Excel i fogli di lavoro possono avere valori con tipi di dati diversi dalle strighe; le celle possono avere impostazioni diverse di caratteri, corpi o colori; le celle possono avere larghezza e altezza variabili; celle adiacenti si possono unire; e si possono incorporare immagini e grafici.

2. Si passa un oggetto `File`, ottenuto da una chiamata a `open()`.
3. Gli oggetti `File` devono essere aperti in modalità lettura binaria ('`rb`') per oggetti `Reader` e in modalità scrittura binaria ('`wb`') per oggetti `Writer`.
4. Il metodo `writerow()`.
5. L'argomento `delimiter` cambia la stringa usata per separare le celle in una riga. L'argomento `lineterminator` modifica la stringa usata per separare le righe.
6. `json.loads()`
7. `json.dumps()`

## Capitolo 15

1. Un momento di riferimento utilizzato da molti programmi di data e ora: è l'1 gennaio 1070, UTC.
2. `time.time()`
3. `time.sleep(5)`
4. Restituisce l'intero più vicino all'argomento passato. Per esempio, `round(2.4)` restituisce 2.
5. Un oggetto `datetime` rappresenta un istante specifico nel tempo. Un oggetto `timedelta` rappresenta un intervallo di tempo.
6. `threadObj = threading.Thread(target=spam)`
7. `threadObj.start()`
8. Fate attenzione che il codice che gira in un thread non legga o scriva le stesse variabili del codice che gira in un altro thread.
9. `subprocess.Popen('c:\\Windows\\System32\\calc.exe')`

## Capitolo 16

1. SMTP e IMAP, rispettivamente.
2. `smtplib.SMTP()`, `smtpObj.ehlo()`, `smptObj.starttls()` e `smtpObj.login()`
3. `imapclient.IMAPClient()` e `imapObj.login()`
4. Una lista di stringhe di parole chiave IMAP, per esempio '`BEFORE <date>`', '`FROM <string>`' o '`SEEN`'
5. Assegnate alla variabile `imaplib._MAXLINE` un valore intero molto grande, per esempio 10000000.
6. Il modulo `pymail` legge le email scaricate.
7. Avrete bisogno di numero SID dell'account Twilio, del numero di authentication token e del vostro numero telefonico Twilio.

## Capitolo 17

1. Un valore RGBA è una tupla di 4 interi, ciascuno dei quali compreso fra 0 e 255. I quattro interi corrispondono alla quantità di rosso, verde, blu e alfa (trasparenza) nel colore.
2. Una chiamata alla funzione `ImageColor.getcolor('CornflowerBlue', 'RGBA')` restituirà (100, 149, 237, 255), il valore RGBA per quel colore.
3. Una tupla di box è un valore tupla contenente quattro interi: la coordinata `x` del bordo sinistro, la coordinata `y` del bordo superiore, la larghezza e l'altezza, rispettivamente.

4. `Image.open('zophie.png')`
5. `imageObj.size` è una tupla di due interi, la larghezza e l'altezza.
6. `imageObj.crop((0, 50, 50, 50))`. Notate che a `crop()` si passa una tupla di box, non quattro argomenti interi distinti.
7. Chiamate il metodo `imageObj.save('new_filename.png')` dell'oggetto `Image`.
8. Il modulo `ImageDraw` contiene il codice per disegnare sulle immagini.
9. Gli oggetti `ImageDraw` hanno metodi per il disegno di forme, come `point()`, `line()` o `rectangle()`. Vengono restituiti passando l'oggetto `Image` alla funzione `ImageDraw.Draw()`.

## Capitolo 18

1. Spostate il mouse nell'angolo superiore sinistro dello schermo, cioè alle coordinate `(0, 0)`.
2. `pyautogui.size()` restituisce una tupla di due interi, la larghezza e l'altezza dello schermo.
3. `pyautogui.position()` restituisce una tupla con due interi per le coordinate `x` e `y` del cursore del mouse.
4. La funzione `moveTo()` sposta il mouse a coordinate assolute sullo schermo, mentre la funzione `moveRel()` sposta il mouse relativamente alla sua posizione corrente.
5. `pyautogui.dragTo()` e `pyautogui.dragRel()`
6. `pyautogui.typewrite('Hello world!')`
7. O passate una lista di stringhe che rappresentano tasti a `pyautogui.typewrite()` (per esempio `'left'`) o passate una singola stringa di tasto a `pyautogui.press()`.
8. `pyautogui.screenshot('screenshot.png')`
9. `pyautogui.PAUSE = 2`