



PREFAZIONE

Se sette anni fa mi aveste detto che mi sarei ritrovato qui seduto a scrivere la prefazione di un libro, tanto meno la prefazione di un libro di programmazione, vi avrei guardato incredulo e probabilmente mi sarei messo a ridere.

Invece sono qui. Sette anni fa ero semplicemente un ingegnere di collaudo con alcune competenze di scripting e una formazione da amministratore di sistema. Non programmavo molto e non avevo alcuna passione per la programmazione, neanche per sogno.

Un giorno, qualcuno che sarebbe diventato presto un mio collega mi parlò di questo “nuovo” linguaggio di “scripting” chiamato Python. Mi disse che era facile da imparare e che nonostante fossi scettico — i programmatori sembravano essere così lontani dal mio “mondo reale” di test, sistemi e utenti — avrei potuto aggiungerlo all’insieme delle mie conoscenze. Andai alla libreria più vicina e comprai il primo libro che trovai.

Il libro che comprai era l’originale *Immersione in Python* di Mark Pilgrim. Devo pensare di non essere la sola persona a poter dire senza esagerare che quel libro ha cambiato la mia vita e la mia carriera per sempre.

Il libro di Mark — la sua passione per Python e per la divulgazione — e il linguaggio stesso hanno fondamentalmente cambiato il mio modo di pensare. Il libro non è stata solo l’ennesima lettura di argomento tecnologico, ma mi ha spinto a programmare e a rappresentare le mie idee in un modo completamente nuovo e sconosciuto. La passione di Mark per il linguaggio mi ha contagiato con una passione scoperta solo in quel momento.

Ora, sette anni dopo, collaboro all’implementazione della libreria standard di Python, sono un membro attivo della comunità e insegno il linguaggio a quante più persone è possibile. Uso Python nel mio tempo libero e lo uso nel mio lavoro. Contribuisco al linguaggio durante i pisolini di mia figlia. *Immersione in Python* — e lo stesso Python — mi hanno cambiato.

Come linguaggio, Python potrebbe non essere il più bello o il più flessibile. Però è pulito, semplice e potente. La sua eleganza sta nella semplicità e nella praticità che gli sono care. La sua flessibilità sta nel mettervi in grado — nel mettere in grado chiunque — di fare qualcosa — qualsiasi cosa — e semplicemente “togliersi di mezzo”.

Lo ripeto da tempo: la bellezza di Python è che scala “verso l’alto” — è utile per qualcuno che voglia solo fare un po’ di matematica o programmare qualcosa di semplice, pur rimanendo utile per i programmatori che vogliono creare sistemi su larga scala, framework per applicazioni web, o siti di condivisione video dal valore di svariati milioni di dollari.

Python ha avuto le sue imperfezioni. Costruire un linguaggio è molto simile, almeno per come la vedo io, a imparare a programmare. È un processo evolutivo in cui dovete costantemente mettere in discussione le decisioni che avete preso ed essere ben disposti a correggere quelle decisioni.

Fondamentalmente, Python 3 è questo: l’ammissione degli errori unita alle successive correzioni per rimuovere alcune delle imperfezioni, magari introducendone delle nuove. Python 3 dimostra una consapevolezza di sé e una volontà di evolvere nelle direzioni più necessarie che non si vedono molto spesso in giro.

Python 3 non ridefinisce tutto il linguaggio Python che conosceste prima, non ne altera le fondamenta né lo invalida di colpo; invece, prende un linguaggio che ha superato la prova del tempo e combattuto tante battaglie e lo migliora in maniera pratica e razionale.

Python 3 non è il punto d’arrivo nell’evoluzione del linguaggio — proprio per niente. Nuove caratteristiche, nuova sintassi e nuove librerie vengono ancora aggiunte e probabilmente verranno ancora aggiunte, modificate e rimosse per tutto il tempo in cui Python continuerà a vivere.

Python 3 è semplicemente una piattaforma più pulita ed evoluta per fare in modo che voi lettori siate produttivi.

Proprio come Python 3, *Immersione in Python 3* è l’evoluzione di una cosa già molto buona in una cosa anche migliore. La passione, l’arguzia e lo stile affascinante di Mark sono rimasti intatti. Il materiale è stato ampliato, migliorato e aggiornato ma, come lo stesso Python 3, la sua essenza è sempre uguale a quella che mi ha donato la passione per la programmazione.

La semplicità di Python è contagiosa. La passione della comunità e la passione con cui il linguaggio è stato creato e viene mantenuto è stupefacente.

Spero che la passione di Mark e lo stesso Python vi siano di ispirazione, come è successo a me. Spero che troverete in Python, e in Python 3, uno strumento tanto pratico e potente quanto lo considerano tale centinaia di migliaia di programmatori e di aziende in tutto il mondo.

Jesse Noller

Programmatore Python

CAPITOLO -1. COSA C'È DI NUOVO IN “IMMERSIONE IN PYTHON 3”

“ Non è da qui che siamo entrati? ”
— Pink Floyd, *The Wall*

ALIAS “IL LIVELLO MENO”

A avete già familiarità con la programmazione in Python? Avete letto l'originale “Immersione in Python”? Magari avete perfino comprato il libro in inglese? (Se è così, grazie!) Siete pronti a saltare il fosso e imparare Python 3? ... Se tutto questo è vero, continuate a leggere. (Se niente di tutto questo è vero, vi converrebbe cominciare dall'inizio.)

Python 3 include uno script chiamato `2to3`. Imparate come funziona. Amatelo. Usatelo. Convertire codice verso Python 3 con 2to3 è una guida di riferimento a tutte le cose che lo strumento `2to3` è in grado di correggere automaticamente. Dato che molte di quelle correzioni sono modifiche alla sintassi, la guida è un buon punto di partenza per imparare molti dei cambiamenti sintattici di Python 3. (Ora `print` è una funzione, ``x`` non funziona, `&c.`)

Caso di studio: convertire `chardet` verso Python 3 documenta il mio tentativo (infine coronato dal successo) di convertire una libreria non banale da Python 2 verso Python 3. Potrebbe esservi d'aiuto; potrebbe non esserlo. Il capitolo ha una curva di apprendimento abbastanza ripida, dato che dovete prima capire più o meno come funziona la libreria in modo da poter poi capire i problemi che sono comparsi e come risolverli. Molti malfunzionamenti coinvolgono le stringhe. Parlando delle quali...

Stringhe. Whew. Da dove cominciare. Python 2 aveva “stringhe” e “stringhe Unicode”. Python 3 ha “byte” e “stringhe”. Cioè, tutte le stringhe ora sono stringhe Unicode, e se volete lavorare con un insieme di byte dovete usare il nuovo tipo `bytes`. Python 3 non effettuerà *mai* una conversione implicita tra stringhe e byte,

quindi se non siete sempre sicuri di cosa state usando i vostri programmi si bloccheranno quasi sicuramente. Leggete il capitolo sulle stringhe per maggiori dettagli.

Il confronto tra byte e stringhe si ripresenta più volte nel corso del libro.

- Nel capitolo File imparerete la differenza tra leggere file in modalità “binaria” e in modalità “testo”. Leggere (e scrivere!) file in modalità testo richiede un parametro encoding. Alcuni metodi dei file di testo contano i caratteri, ma altri metodi contano i byte. Se il vostro programma presume che un carattere sia uguale a un byte, si *bloccherà* sui caratteri multibyte.
- Nel capitolo Servizi web HTTP il modulo `httplib2` preleva intestazioni e dati via HTTP. Le intestazioni HTTP vengono restituite come stringhe, ma il corpo di una risposta HTTP viene restituito sotto forma di byte.
- Nel capitolo Serializzare oggetti Python imparerete perché il modulo `pickle` in Python 3 definisce un nuovo formato di dati che non è compatibile all’indietro con Python 2. (Suggerimento: è a causa di byte e stringhe.) Inoltre, Python 3 supporta la serializzazione da e verso il formato JSON, che nemmeno possiede un tipo di dato bytes. Vi mostrerò come aggirare questa limitazione.
- Il capitolo Caso di studio: convertire chardet verso Python 3 è un macello totale di byte e stringhe.

Anche se non siete interessati a Unicode (oh, ma lo sarete), vorrete sapere come formattare le stringhe in Python 3, che è un’operazione completamente diversa rispetto a Python 2.

Gli iteratori sono ovunque in Python 3, e li capisco molto meglio rispetto a quando ho scritto “Immersione in Python” cinque anni fa. Anche voi avete bisogno di capirli, perché molte funzioni che erano solite restituire liste in Python 2 ora restituiranno iteratori in Python 3. Come minimo, dovrete leggere la seconda metà del capitolo sugli iteratori e la seconda metà del capitolo sull’uso avanzato degli iteratori.

A grande richiesta ho aggiunto un’appendice sui nomi dei metodi speciali, che è più o meno simile al capitolo intitolato “Modello dei dati” nella documentazione di Python ma è condita da osservazioni ironiche e pungenti.

Al tempo in cui stavo scrivendo “Immersione in Python”, tutte le librerie XML disponibili facevano schifo. Poi Fredrik Lundh ha implementato ElementTree, che non fa per niente schifo. Gli dèi di Python hanno saggiamente incorporato ElementTree nella libreria standard e ora ElementTree forma le basi per il mio nuovo capitolo su XML. Le vecchie librerie per il riconoscimento di XML sono ancora in giro, ma dovrete evitarle, perché fanno schifo!

Un'altra novità di Python — non nel linguaggio ma nella comunità — è la comparsa di archivi di codice come il Python Package Index (PyPI, Indice dei Pacchetti Python). Python viene distribuito con alcuni strumenti utili per impacchettare il vostro codice in formati standard e distribuire quei pacchetti su PyPI. Leggete il capitolo Distribuire librerie Python per conoscere i dettagli.

CAPITOLO 0. INSTALLARE PYTHON

“ *Tempora mutantur nos et mutamur in illis. (I tempi cambiano e noi cambiamo con loro.)* ”
— *antico proverbio Romano*

0.1. IMMERSIONE!

Benvenuti in Python 3. Immergiamoci. La prima cosa che dovrete fare con Python è installarlo. Oppure no?

0.2. QUALE PYTHON VI SERVE?

Se state usando un account su un server remoto, il vostro ISP potrebbe avere già installato Python 3. Se state usando Linux a casa, anche in questo caso potreste già avere Python 3. La maggior parte delle distribuzioni GNU/Linux popolari includono Python 2 nella propria installazione predefinita; il numero di distribuzioni che includono anche Python 3 è limitato, ma in costante aumento. Mac OS X include una versione di Python 2 a riga di comando, ma al momento della scrittura non include Python 3. Microsoft Windows non include alcuna versione di Python. Ma non disperate! Potete aprirvi la strada verso l'installazione di Python a colpi di mouse, a prescindere da quale sistema operativo usate.

Il modo più facile di controllare se avete Python 3 sul vostro sistema Linux o Mac OS X è quello di ricorrere alla riga di comando. Su Linux, cercate un programma chiamato Terminale nel vostro menu Applicazioni. (Potrebbe trovarsi in un sottomenu come Accessori o Strumenti di sistema.) Su Mac OS X, c'è un'applicazione chiamata Terminal.app nella vostra cartella /Applications/Utilities/.

Una volta che vi trovate al prompt della riga di comando, digitate semplicemente `python3` (tutto in minuscolo, senza spazi) e guardate cosa succede. Sul mio sistema Linux di casa Python 3 è già installato, così questo comando mi fa entrare nella *shell interattiva di Python*.

```
mark@atlantis:~$ python3
Python 3.0.1+ (r301:69556, Apr 15 2009, 17:25:52)
[GCC 4.3.3] on linux2
Digitate "help", "copyright", "credits" o "license" per maggiori informazioni.
>>>
```

(Digitate `exit()` e premete INVIO per uscire dalla shell interattiva di Python.)

Anche il mio fornitore di servizi web usa Linux e offre l'accesso alla riga di comando, ma Python 3 non è installato sul mio server. (Buu!)

```
mark@manganese:~$ python3
bash: python3: comando non trovato
```

Quindi, tornando alla domanda con cui questa sezione è cominciata: “Quale Python vi serve?” Qualunque sia quello già installato sul computer che avete.

[Continuate a leggere per le istruzioni di installazione su Windows, oppure saltate a Installare su Mac OS X, Installare su Ubuntu Linux, o Installare su altre piattaforme.]



0.3. INSTALLARE SU MICROSOFT WINDOWS

Oggigiorno Windows viene distribuito per due architetture: 32-bit e 64-bit. Naturalmente, ci sono molte *versioni* differenti di Windows — XP, Vista, Windows 7 — ma Python funziona su tutte. La distinzione più importante è quella tra 32-bit e 64-bit. Se non avete idea di quale sia l'architettura che state usando, probabilmente è a 32-bit.

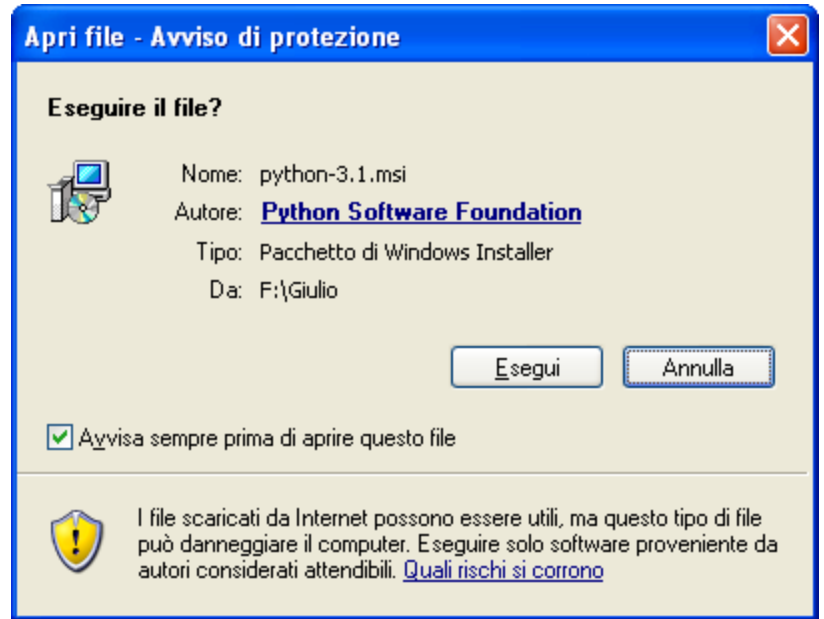
Visitate python.org/download/ e scaricate il programma di installazione di Python 3 per Windows appropriato per la vostra architettura. Le vostre scelte somiglieranno a queste:

- **Python 3.1 Windows installer** (eseguibile Windows — non include i sorgenti)
- **Python 3.1 Windows AMD64 installer** (eseguibile Windows AMD64 — non include i sorgenti)

Preferisco evitare di fornirvi i collegamenti diretti per scaricare i programmi di installazione, perché aggiornamenti minori di Python vengono rilasciati in ogni momento e non voglio avere la responsabilità di farvi perdere qualche aggiornamento importante. Dovreste sempre installare la versione più recente di Python 3.x a meno che non abbiate qualche oscura ragione per non farlo.

Fate doppio click sul file .msi dopo aver finito di scaricarlo. Windows vi presenterà un avvertimento di sicurezza, dato che state per lanciare un file eseguibile. Il programma di installazione ufficiale di Python è firmato digitalmente dalla Python Software Foundation, la società no-profit che sovrintende allo sviluppo di Python. Diffidate dalle imitazioni!

Cliccate sul bottone Esegui per lanciare il programma di installazione di Python 3.



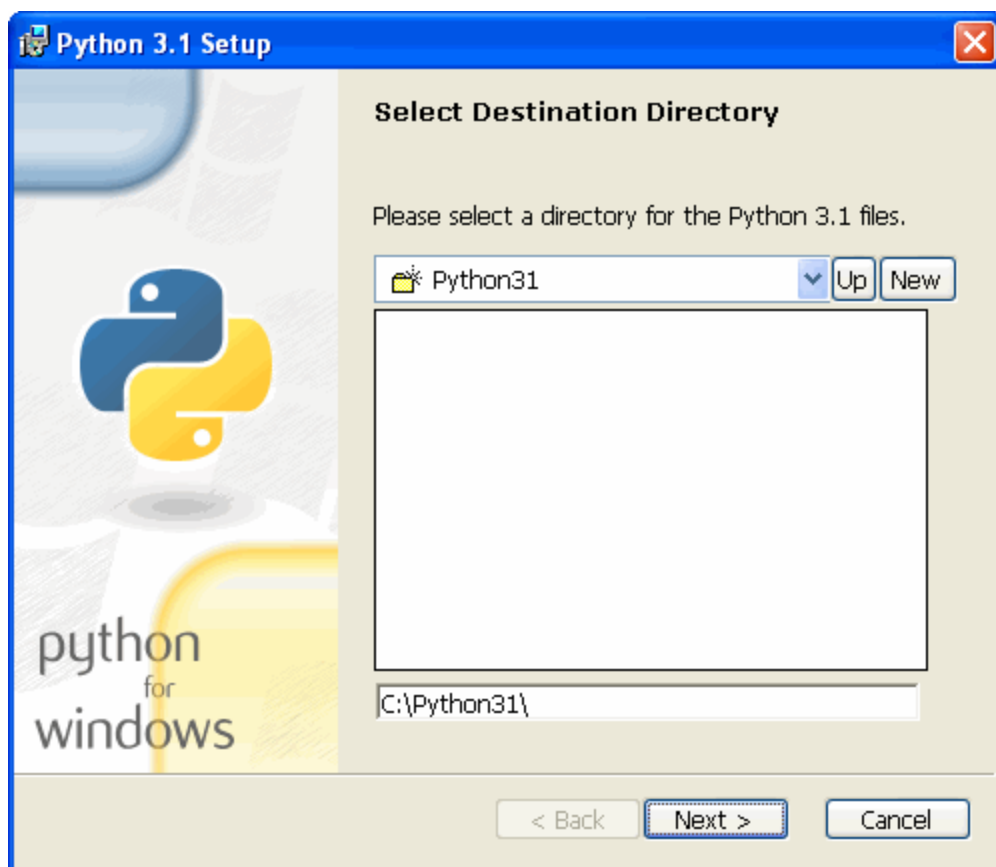
Come prima cosa, il programma di installazione vi chiederà se volete installare Python 3 per tutti gli utenti o solo per voi. La scelta predefinita è “install for all users” (cioè “installa per tutti gli utenti”), che è quella migliore a meno che non abbiate una buona ragione per scegliere altrimenti. (Una delle possibili ragioni per cui potreste preferire “install just for me”, cioè “installa solo per me”, è che state installando Python sul computer della vostra azienda e non avete i

diritti di amministrazione sul vostro account Windows. Ma allora perché state installando Python senza il permesso dell'amministratore di Windows della vostra azienda? Non cacciatemi in questi guai!)

Cliccate sul bottone Next per confermare la vostra scelta sul tipo di installazione.



Successivamente, il programma di installazione vi chiederà di scegliere una directory di destinazione. La scelta predefinita per tutte le versioni di Python 3.1.x è C:\Python31\, che dovrebbe andare bene per la maggior parte degli utenti a meno che non ci sia una ragione particolare per cambiarla. Se usate una partizione separata per installare le applicazioni, potete selezionarla usando i controlli incorporati, oppure potete semplicemente digitare il percorso nel campo di testo sottostante. Non siete costretti a installare Python sul disco C:, ma potete installarlo su qualsiasi disco e in qualsiasi cartella.



Cliccate sul bottone Next per confermare la vostra scelta sulla directory di destinazione.

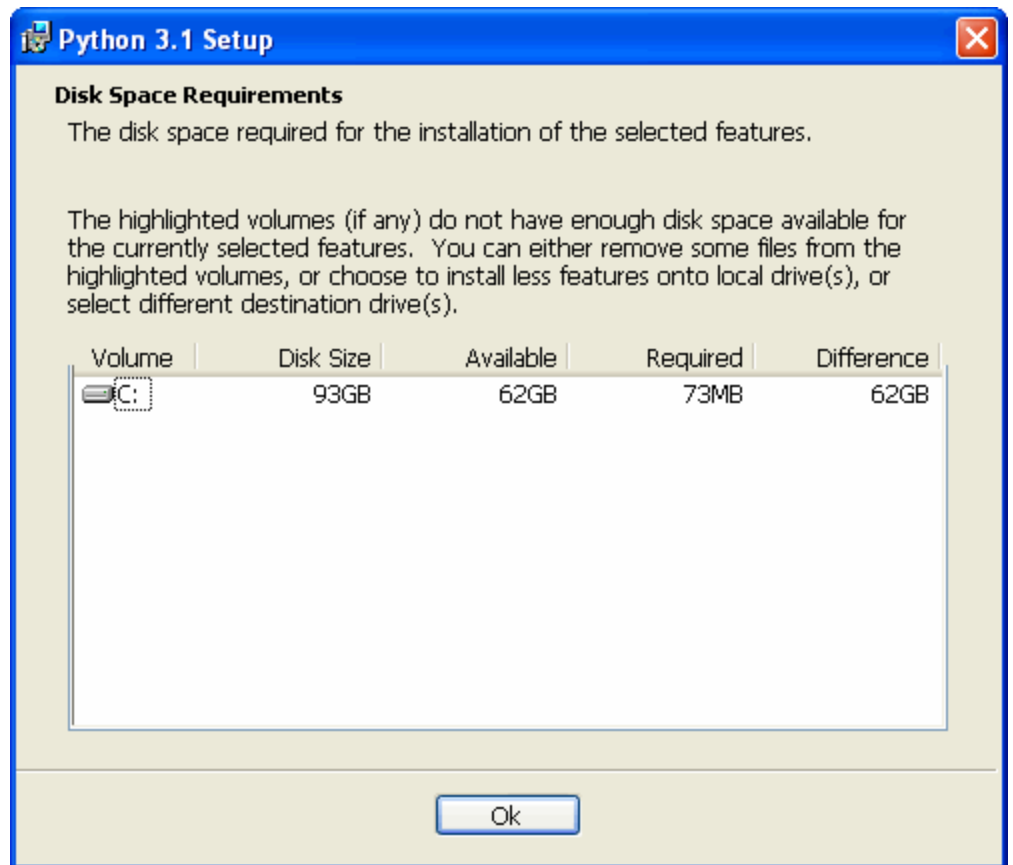
La schermata seguente sembra complicata, ma in realtà non lo è. Come accade in molti programmi di installazione, avete la possibilità di non installare ogni singolo componente di Python 3. Se lo spazio su disco è limitato, potete escludere alcuni componenti.

- **Register Extensions** (estensioni del registro) vi permette di fare doppio click sugli script Python (i file .py) per eseguirli. Raccomandato ma non obbligatorio. (Questa opzione non richiede spazio su disco, quindi non ha molto senso escluderla.)
- **Tcl/Tk** è la libreria grafica usata dalla Shell Python che userete in tutto questo libro. Raccomando vivamente di tenere questa opzione selezionata.
- **Documentation** (documentazione) installa un file di aiuto che contiene molte delle informazioni che si trovano su docs.python.org. Raccomandata se avete una connessione in dial-up o un accesso a Internet limitato.
- **Utility Scripts** (programmi di utilità) include lo script 2to3.py che imparerete a conoscere più avanti in questo libro. Questo script è necessario se volete imparare a convertire il codice Python 2 esistente verso Python 3. Se non avete codice Python 2 esistente, potete tralasciare questa opzione.
- **Test Suite** (serie di test) è una collezione di script usata per collaudare l'interprete Python. Non la useremo in questo libro, né personalmente l'ho mai usata nelle attività di programmazione in Python. Completamente opzionale.



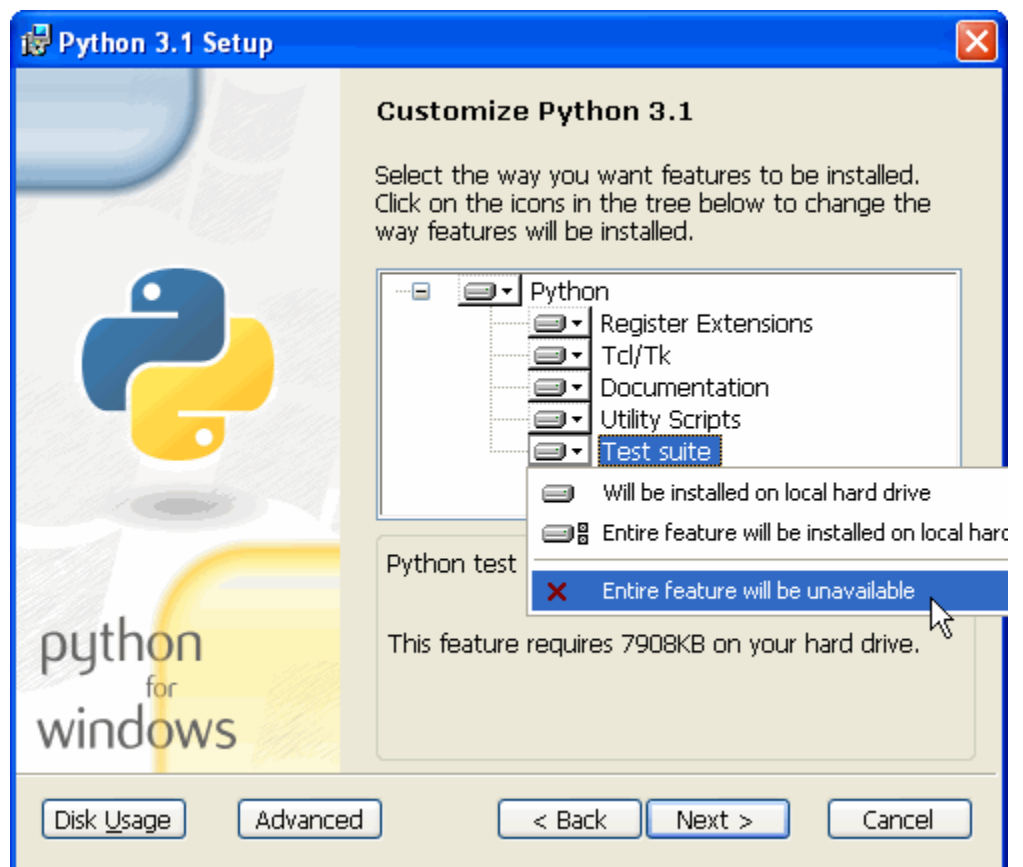
Se non siete sicuri di quanto spazio libero avete su disco, cliccate sul bottone Disk Usage. Il programma di installazione elencherà i vostri dischi e calcolerà sia quanto spazio è disponibile su ognuno sia quanto spazio rimarrebbe dopo l'installazione.

Cliccate sul bottone OK per ritornare alla schermata "Customizing Python" (cioè "Personalizzare Python").

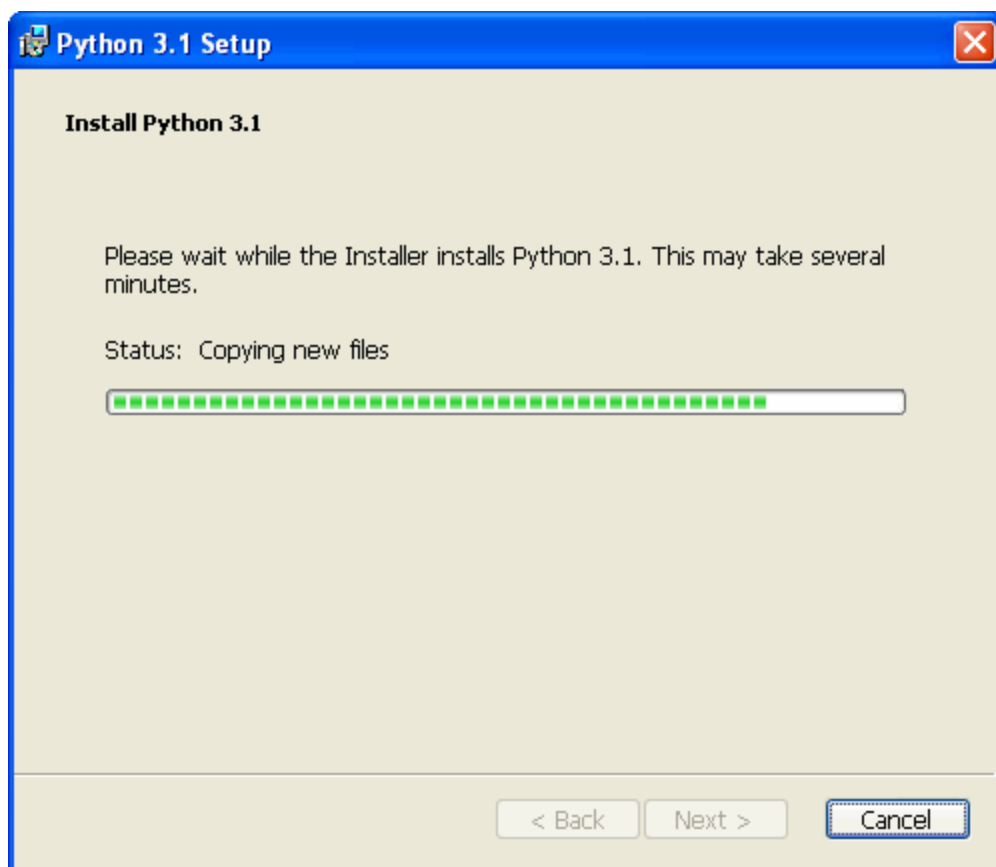


Se decidete di escludere un'opzione, cliccate sul bottone alla sinistra dell'opzione e selezionate "Entire feature will be unavailable" (cioè "L'intera funzione non sarà disponibile") dalla lista a cascata che comparirà. Per esempio, escludendo i test risparmierete l'incredibile spazio su disco di 7908 KB.

Cliccate sul bottone Next per confermare la vostra scelta sulle opzioni.



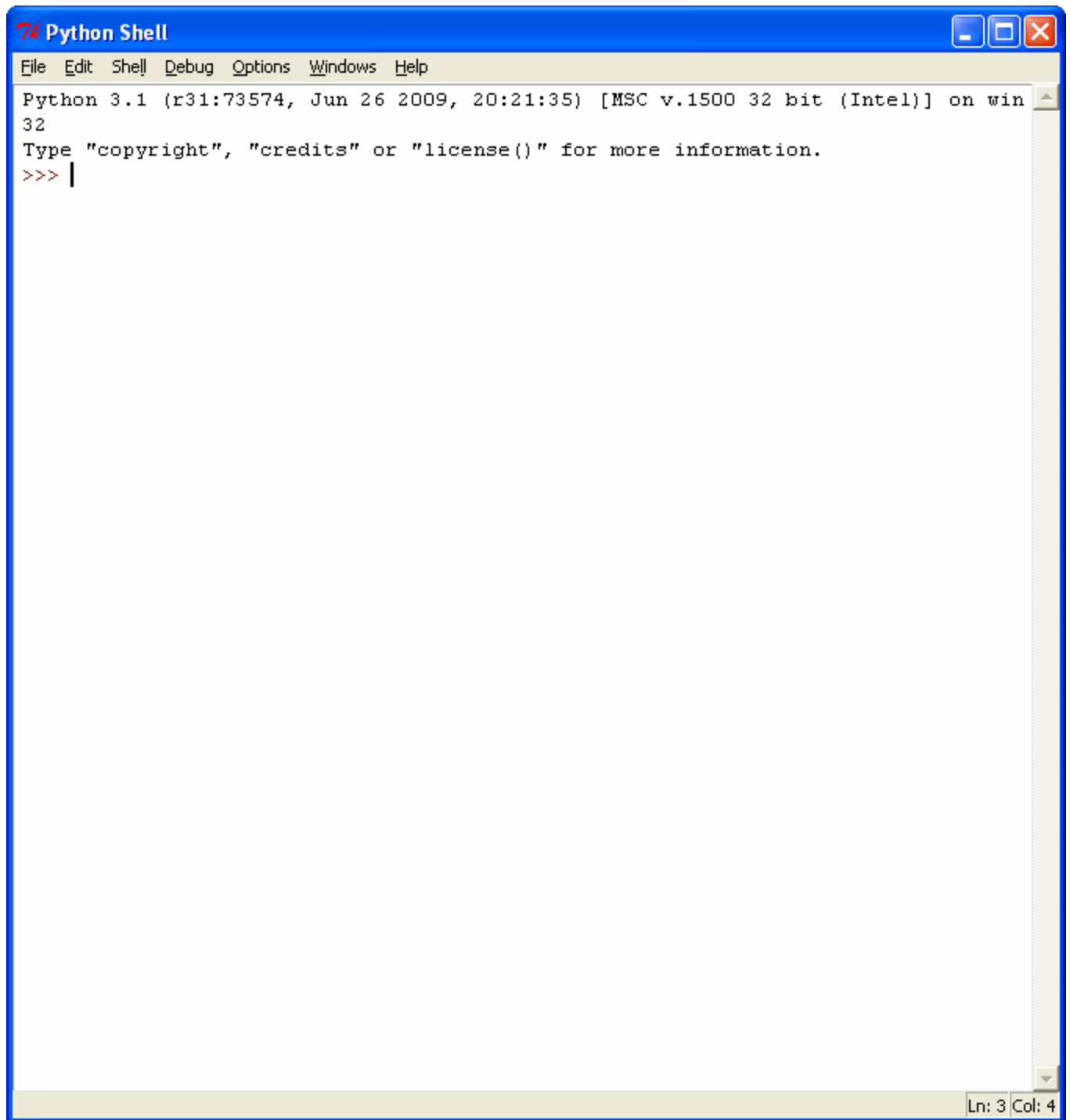
Il programma di installazione copierà tutti i file necessari nella directory di destinazione che avete scelto. (Questa operazione avviene così velocemente che ho dovuto fare ben tre tentativi prima di riuscire a catturarne un'immagine!)



Cliccate sul bottone Finish per uscire dal programma di installazione.



Nel



vostro menu Start dovrebbe esserci un nuovo elemento chiamato Python 3.1. Al suo interno, c'è un programma chiamato IDLE. Selezionate questo elemento per lanciare la Shell Python interattiva.

[Saltate a usare la Shell Python]

*
**

0.4. INSTALLARE SU MAC OS X

Tutti i moderni computer Macintosh usano microprocessori Intel (come la maggior parte dei PC Windows). I Mac più vecchi usano microprocessori PowerPC. Non avete bisogno di capire la differenza, perché c'è un solo programma di installazione di Python per tutti i Mac.

Visitate python.org/download/ e scaricate il programma di installazione per Mac. Avrà un nome simile a **Python 3.1 Mac Installer Disk Image**, anche se il numero di versione potrebbe essere diverso. Assicuratevi di scaricare la versione 3.x, non la versione 2.x.



Il vostro browser dovrebbe montare automaticamente l'immagine del disco e aprire una finestra del Finder per mostrarvene i contenuti. (Se questo non accade, dovrete trovare l'immagine del disco nella cartella in cui l'avete scaricata e fare doppio clic su di essa per montarla. Il suo nome dovrebbe essere qualcosa di simile a python-3.1.dmg.) L'immagine del disco contiene un certo numero di file di testo (Build.txt, License.txt, ReadMe.txt) e il pacchetto di installazione vero e proprio, chiamato Python.mpkg.

Fate doppio clic sul pacchetto di installazione Python.mpkg per lanciare il programma di installazione di Python per Mac.

La prima schermata del programma di installazione vi presenta una breve descrizione di Python, poi vi rimanda al file `ReadMe.txt` (che non avete letto, giusto?) per maggiori dettagli.



Cliccate sul bottone Continue per proseguire.

La pagina
successiva
contiene



effettivamente alcune importanti informazioni: Python richiede Mac OS X 10.3 o una versione più recente. Se state ancora usando Mac OS X 10.2, dovrete davvero aggiornarlo. Apple non rilascia più aggiornamenti di sicurezza per il vostro sistema operativo e il vostro computer sarà probabilmente a rischio se doveste mai collegarvi in rete. In più, non potete usare Python 3.

Cliccate sul bottone Continue per avanzare.

Come tutti i migliori programmi di installazione, il programma di installazione di Python vi mostra il contratto di licenza software. Python è open source e la sua licenza è



approvata dalla Open Source Initiative. Python ha avuto un certo numero di proprietari e sponsor durante la sua storia, ognuno dei quali ha lasciato il suo marchio sulla licenza software. Ma il risultato finale è questo: Python è open source e potete usarlo su qualsiasi piattaforma, per qualsiasi scopo, senza pagare nulla e senza avere alcun obbligo di reciprocità.

Cliccare sul bottone Continue ancora una volta.

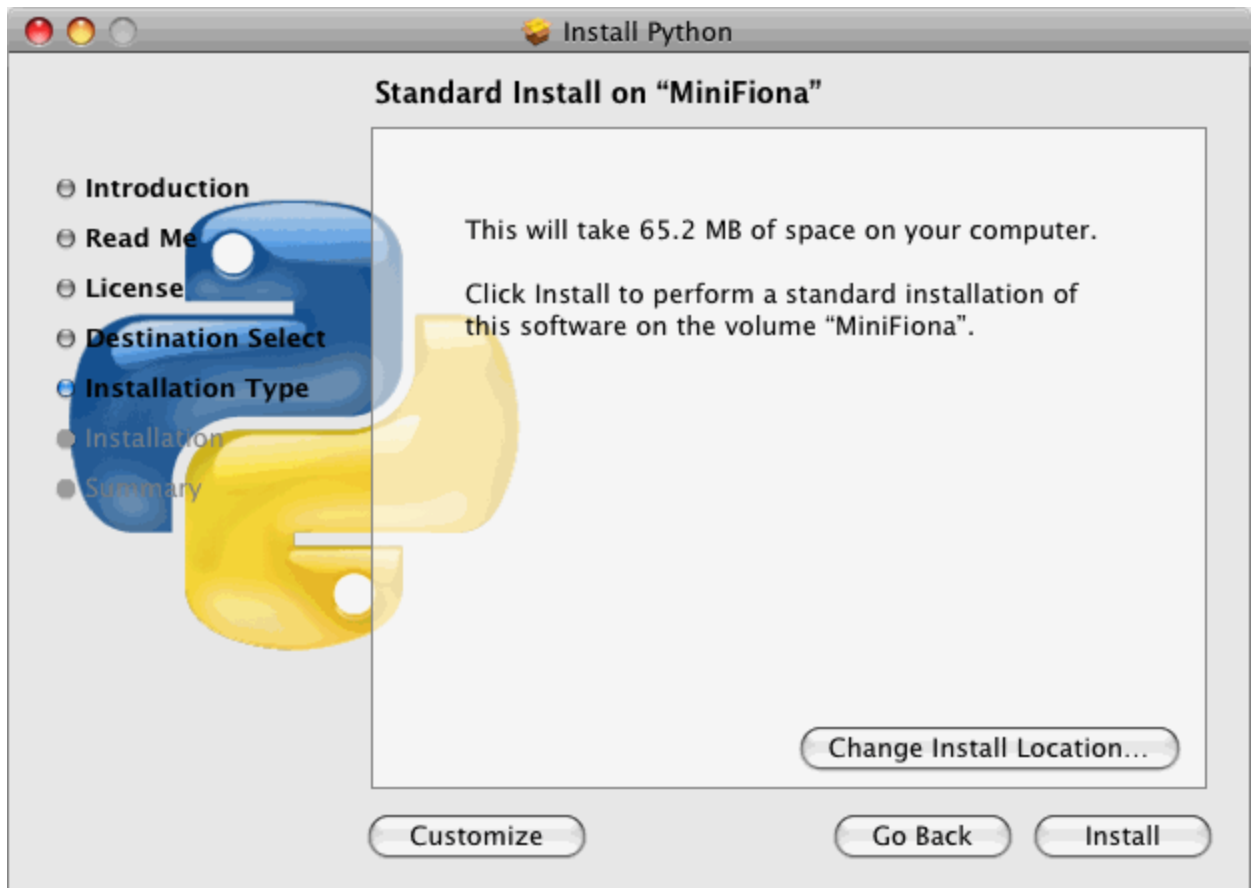
A causa dei cavilli del framework di installazione standard di Apple, dovete “accettare” i termini della licenza software per poter completare



l'installazione. Dato che Python è open source, in realtà state “accettando” che la licenza vi garantisca diritti aggiuntivi piuttosto che privarvene.

Cliccate sul bottone Agree per continuare.

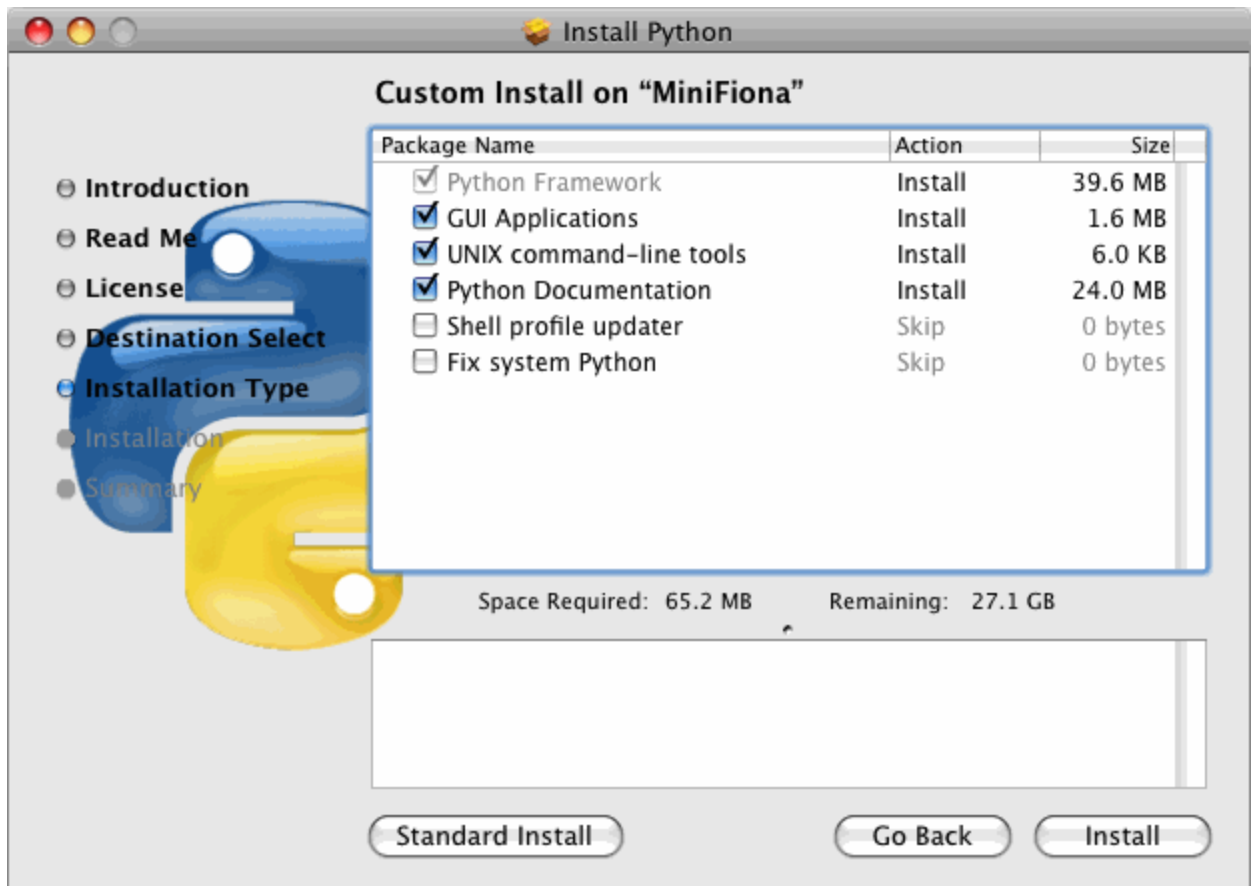
La schermata successiva vi permette di cambiare il percorso di installazione. **Dovete** installare Python sul disco di avvio, ma a causa delle limitazioni del programma di installazione



questo non viene imposto. A dire il vero, non ho mai avuto necessità di cambiare il percorso di installazione.

Da questa schermata potete anche personalizzare l'installazione per escludere alcune funzionalità. Se volete fare questo, cliccate sul bottone *Customize*, altrimenti cliccate sul bottone *Install*.

Se scegliete

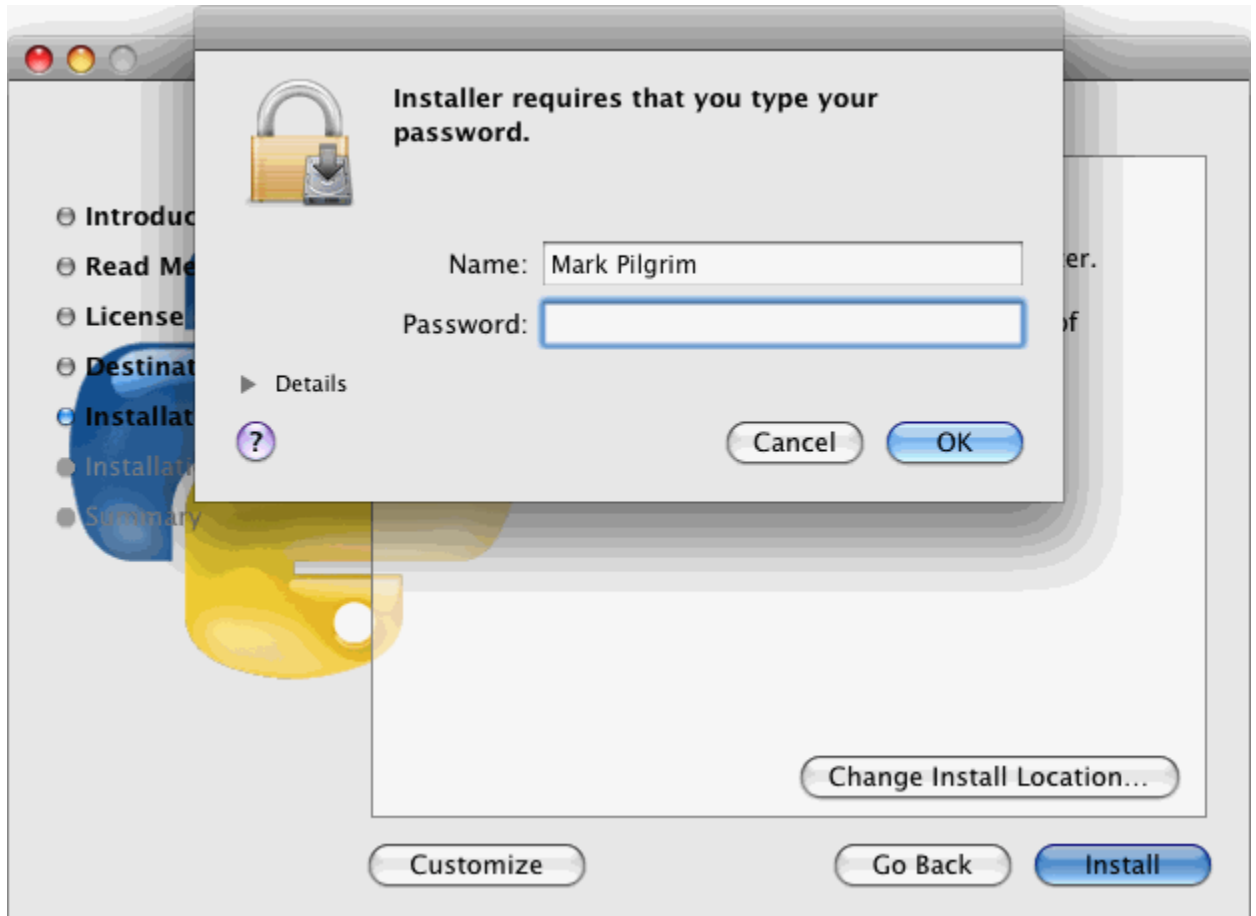


un'installazione personalizzata, il programma di installazione vi mostrerà la seguente lista di funzionalità:

- **Python Framework** (framework Python). Questo è il cuore di Python ed è sia selezionato che disabilitato perché deve per forza essere installato.
- **GUI Applications** (applicazioni grafiche) comprende IDLE, la Shell Python grafica che userete in tutto questo libro. Raccomando vivamente di tenere questa opzione selezionata.
- **UNIX command-line tools** (strumenti UNIX a riga di comando) include l'applicazione python3 a riga di comando. Raccomando vivamente di tenere anche questa opzione.
- **Python Documentation** (documentazione Python) contiene molte delle informazioni che si trovano su docs.python.org. Raccomandata se avete una connessione in dial-up o un accesso a Internet limitato.
- **Shell profile updater** (programma di aggiornamento del profilo di shell) controlla se aggiornare il vostro profilo di shell (usato in Terminal.app) per assicurarsi che questa versione di Python sia quella contenuta nel percorso di ricerca della vostra shell. Probabilmente non avete bisogno di modificare questa impostazione.
- **Fix system Python** (correzione per l'interprete Python di sistema) non dovrebbe essere modificato. (Dice al vostro Mac di usare Python 3 come interprete Python predefinito per tutti gli script, compresi gli script di sistema installati da Apple. Questo sarebbe un vero disastro, dato che molti di quegli script sono stati scritti per Python 2 e non verrebbero eseguiti correttamente da Python 3.)

Cliccate sul bottone Install per continuare.

Dato che i framework di sistema e gli eseguibili vengono installati in /usr/local/bin/, il programma di installazione vi chiederà una password di



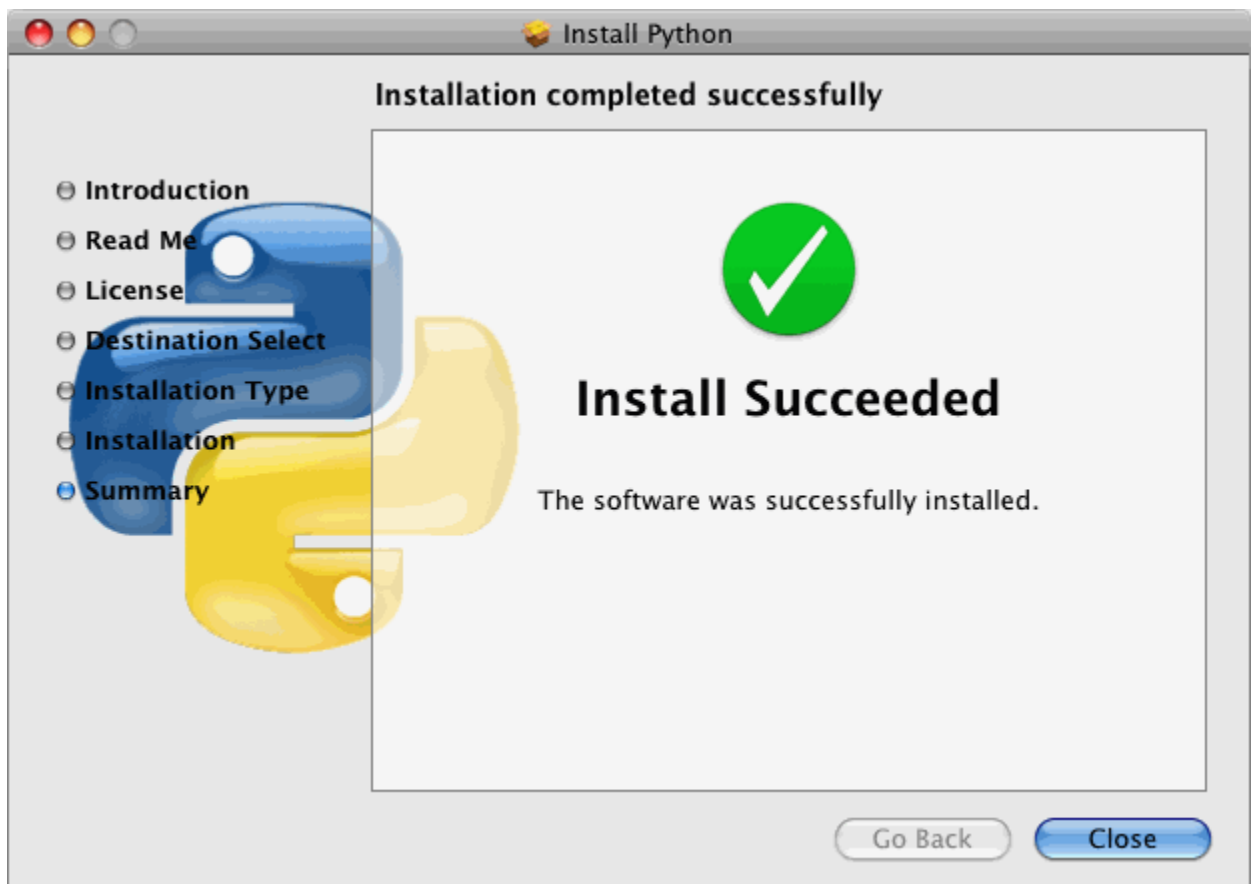
amministrazione. Non c'è alcun modo di installare Python per Mac senza privilegi di amministrazione.

Cliccate sul bottone OK per cominciare l'installazione.

Il programma di installazione mostrerà un indicatore di progresso mentre installa le funzionalità che avete selezionato.



Supponendo che tutto sia andato bene, il programma di installazione



vi mostrerà un grande segno di spunta verde per comunicarvi che l'installazione è stata completata con successo.

Cliccate sul bottone Close per uscire dal programma di installazione.

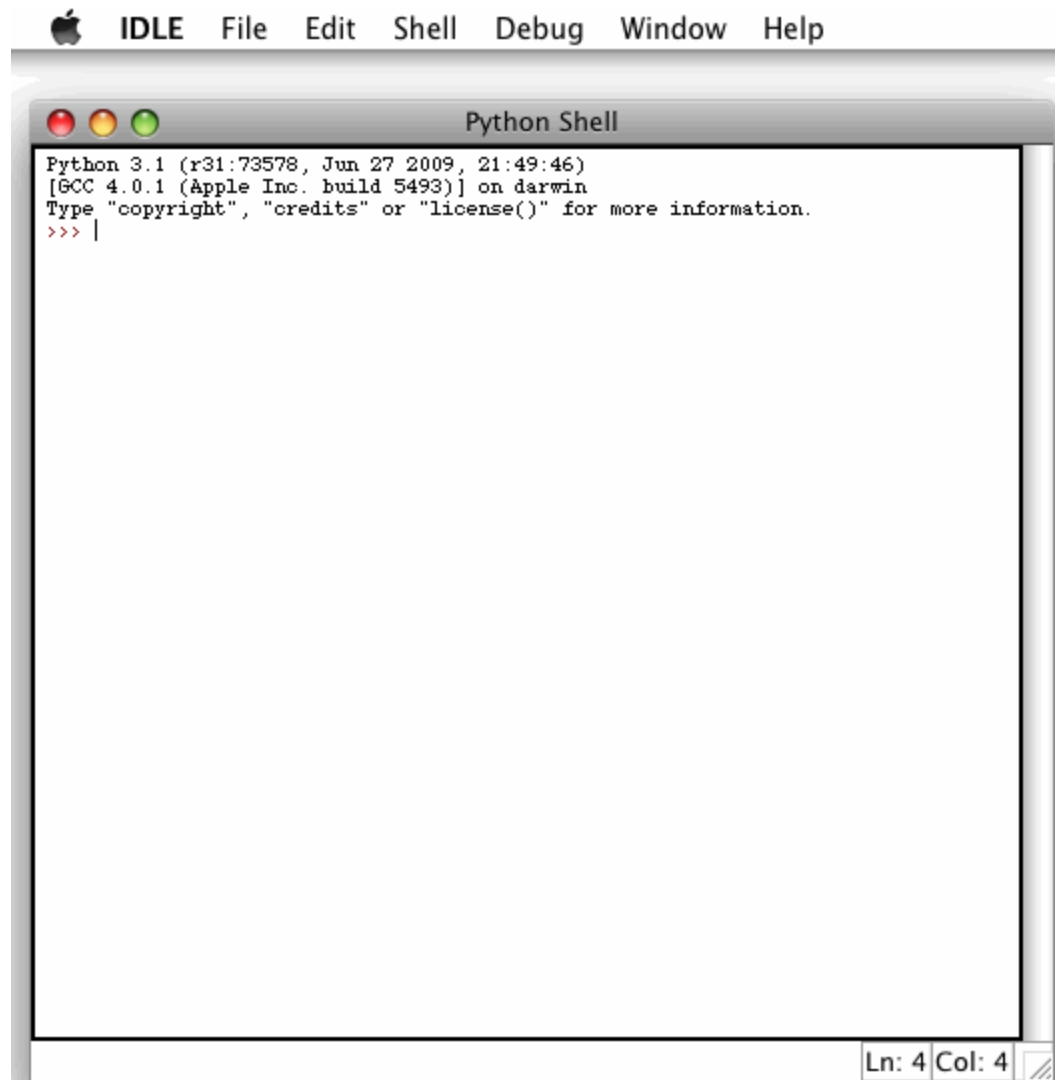
Supponendo che non abbiate cambiato il percorso di installazione, potete trovare i file appena installati nella cartella Python 3.1 all'interno della vostra cartella /Applications. L'elemento più importante è IDLE, la Shell Python grafica.

Fate doppio clic su IDLE per lanciare la Shell Python.



La Shell Python è il luogo in cui passerete la maggior parte del vostro tempo durante l'esplorazione del linguaggio. Gli esempi in questo libro presumiranno che sappiate usare la Shell Python.

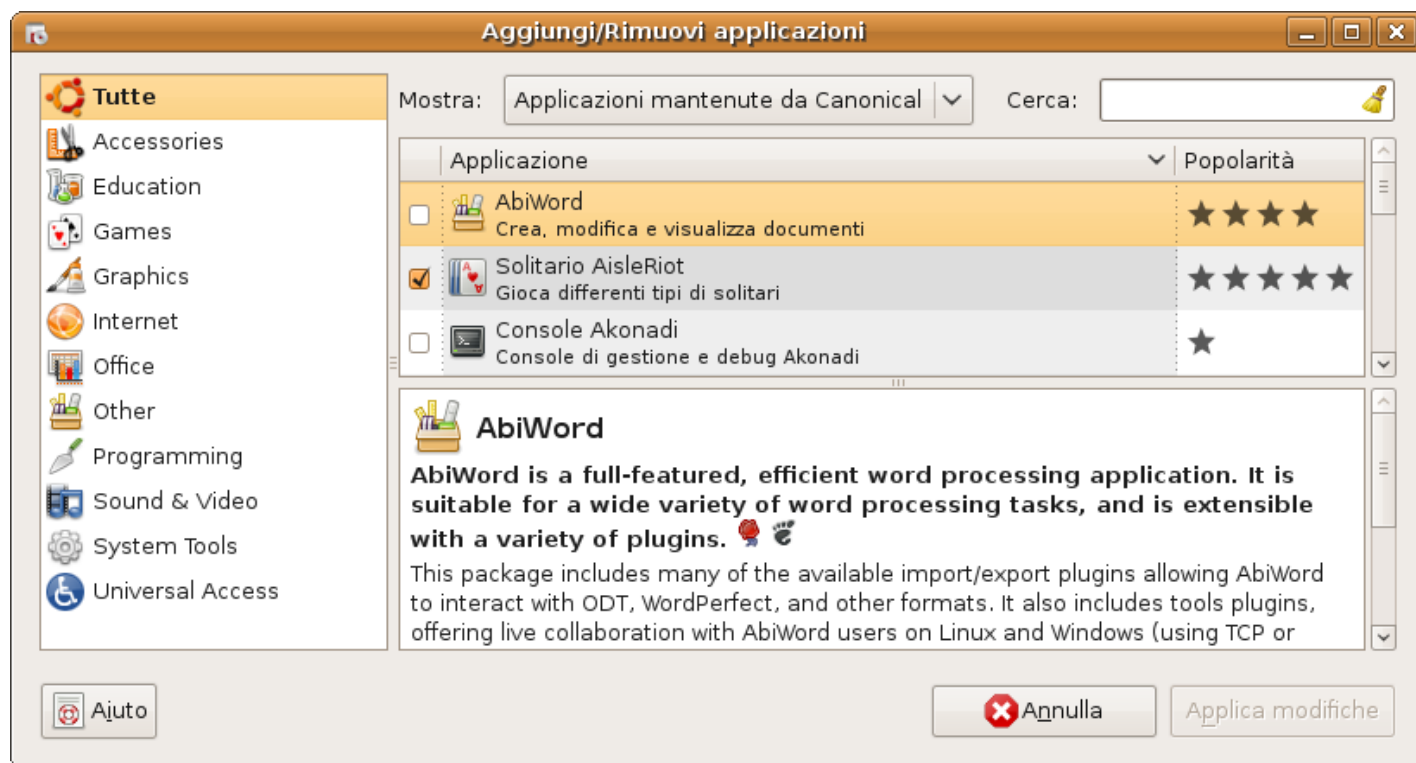
[Saltate a [usare la Shell Python](#)]



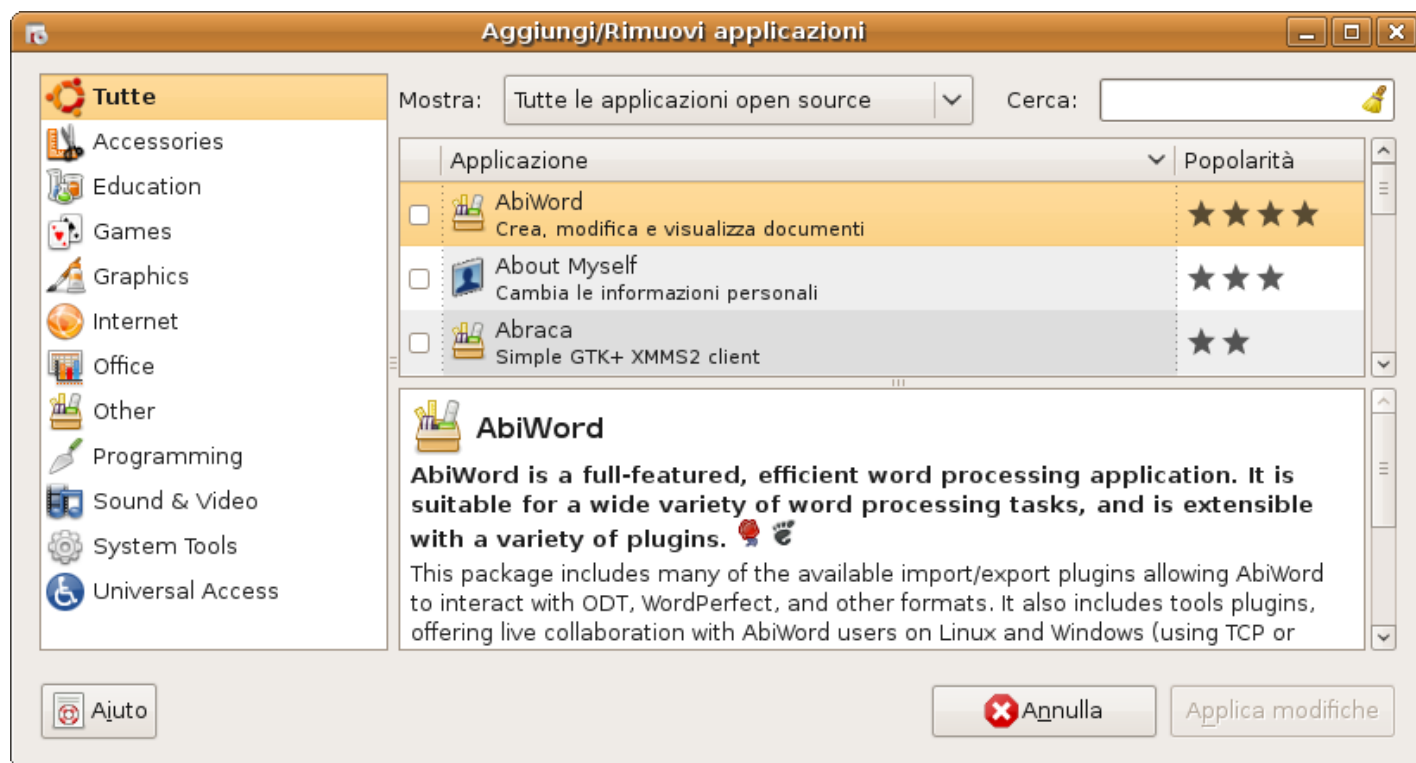
*
**

0.5. INSTALLARE SU UBUNTU LINUX

Le moderne distribuzioni Linux sono supportate da enormi archivi di applicazioni precompilate pronte da installare. I dettagli esatti variano a seconda della distribuzione. Su Ubuntu Linux, il modo più facile per installare Python 3 è attraverso l'applicazione Aggiungi/Rimuovi nel vostro menu Applicazioni.



Quando lanciate per la prima volta l'applicazione Aggiungi/Rimuovi, vi verrà mostrata una lista di applicazioni preselezionate raggruppate in diverse categorie. Alcune sono già installate, altre no. Dato che l'archivio contiene più di 10.000 applicazioni, ci sono diversi filtri che potete applicare per vedere porzioni ridotte dell'archivio. Il filtro predefinito è "Applicazioni mantenute da Canonical", che è un piccolo sottoinsieme del numero totale di applicazioni ufficialmente supportate da Canonical, l'azienda che crea e mantiene la distribuzione Ubuntu.



Python 3 non è mantenuto da Canonical, quindi il primo passo da fare è scorrere il menu dei filtri e selezionare “Tutte le applicazioni open source”.



Una volta che avete espanso il filtro per includere tutte le applicazioni open source, usate la casella Cerca immediatamente a destra del menu dei filtri per cercare python 3.



Ora la lista delle applicazioni si è ristretta alle sole che corrispondono a python 3. Selezionerete due pacchetti. Il primo è Python (v. 3.0), che contiene l'interprete Python.



Il secondo pacchetto che volete è immediatamente sopra: IDLE (utilizza Python-3.0). Questa è la Shell Python grafica che userete in questo libro.

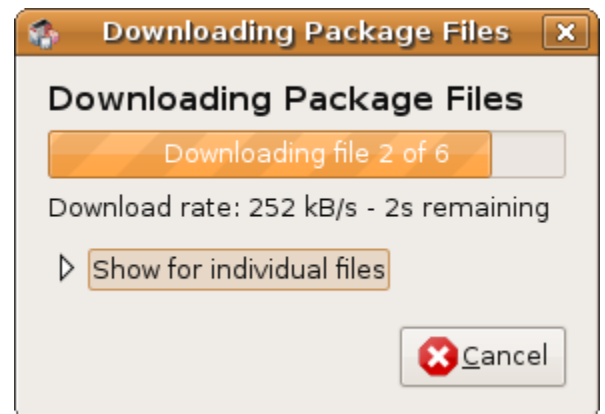
Dopo aver selezionato questi due pacchetti, cliccate sul bottone *Applica modifiche* per continuare.

Il programma di gestione dei pacchetti vi chiederà di confermare che volete aggiungere sia IDLE (utilizza Python-3.0) sia Python (v. 3.0).

Cliccate sul bottone *Applica* per continuare.



Il programma di gestione dei pacchetti vi mostrerà un indicatore di progresso mentre scarica i pacchetti necessari dall'archivio online di Canonical.

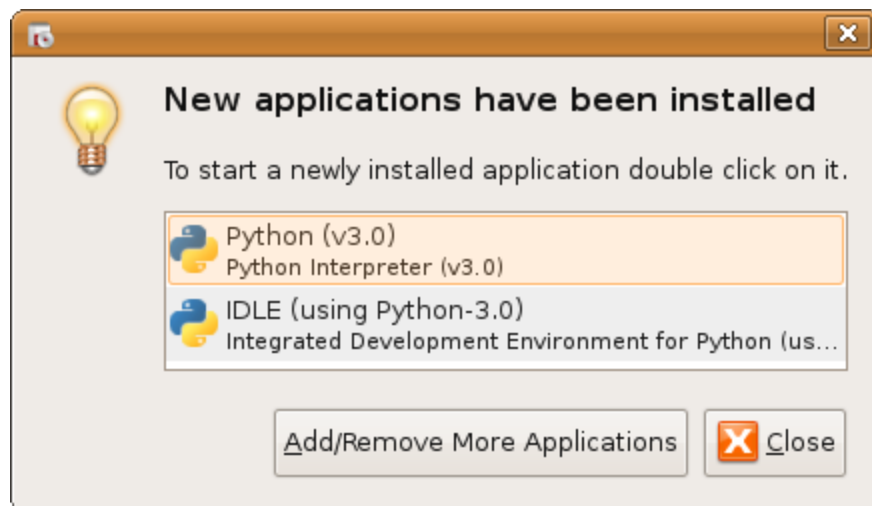


Una volta che i pacchetti sono stati scaricati, il programma di gestione dei pacchetti comincerà automaticamente a installarli.

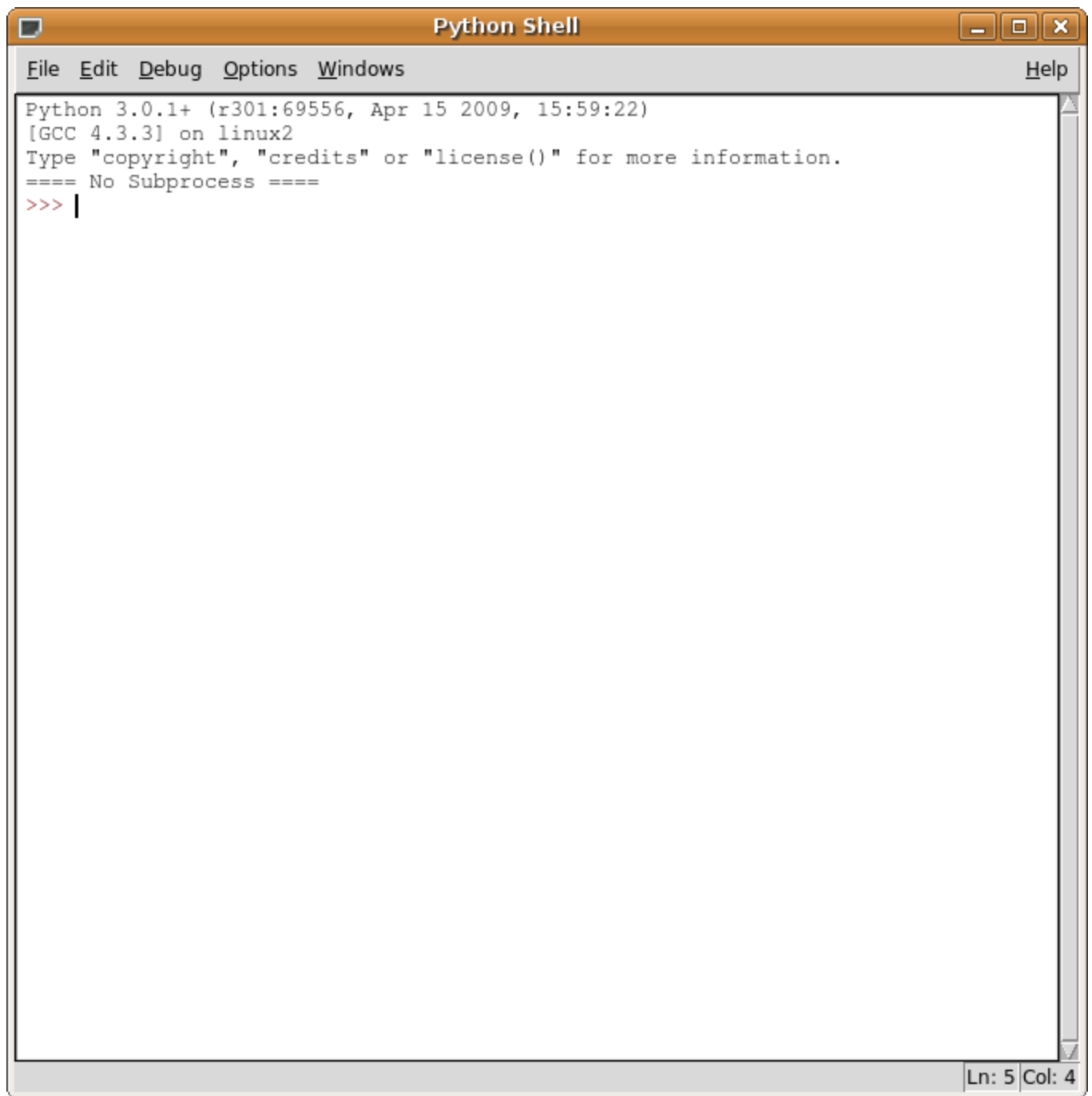


Se tutto è andato bene, il programma di gestione dei pacchetti confermerà che entrambi i pacchetti sono stati installati con successo. Da qui, potete fare doppio clic su IDLE per lanciare la Shell Python, oppure potete cliccare sul bottone Close per uscire dal programma di gestione dei pacchetti.

Potete sempre rieseguire la Shell Python andando nel vostro menu Applicazioni, poi nel sottomenu Programmazione e selezionando la voce IDLE.



La
Shell



```
Python 3.0.1+ (r301:69556, Apr 15 2009, 15:59:22)
[GCC 4.3.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>> |
```

Python è il luogo in cui passerete la maggior parte del vostro tempo durante l'esplorazione del linguaggio. Gli esempi in questo libro presumono che sappiate usare la Shell Python.

[Saltate a [usare la Shell Python](#)]

*
**

0.6. INSTALLARE SU ALTRE PIATTAFORME

Python 3 è disponibile su un certo numero di piattaforme differenti. In particolare, è disponibile in pratica su ogni distribuzione basata su Solaris, Linux e BSD. Per esempio, RedHat Linux usa il gestore di pacchetti yum, FreeBSD ha la sua collezione di pacchetti e conversioni, SUSE ha zypper, e Solaris ha pkgadd e compagna. Una ricerca veloce sul web per Python 3 + *il vostro sistema operativo* dovrebbe indicarvi se esiste un pacchetto di Python 3 disponibile e, in questo caso, come installarlo.



0.7. USARE LA SHELL PYTHON

La Shell Python è il luogo in cui potete esplorare la sintassi Python, ottenere aiuto interattivo sui comandi e fare il debug di piccoli programmi. La Shell Python grafica (chiamata IDLE) contiene anche un editor di testo decente che supporta la colorazione della sintassi Python e si integra con la Shell Python. Se non avete ancora un vostro editor preferito, dovrete provare a usare IDLE.

Cominciamo dall'inizio. La Shell Python in sé è un meraviglioso campo di gioco interattivo. In questo libro, vedrete esempi come questo:

```
>>> 1 + 1  
2
```

Le tre parentesi angolari, >>>, denotano il prompt della Shell Python. Quella parte serve solo a farvi sapere che questo esempio è pensato per essere seguito nella Shell Python, quindi non va digitata.

1 + 1 è la parte che dovete digitare. Potete digitare qualsiasi espressione Python valida o qualsiasi comando nella Shell Python. Non siate timidi, non vi morderà! Il peggio che potrebbe accadervi è ottenere un messaggio di errore. I comandi vengono eseguiti immediatamente (non appena premete INVIO); le espressioni vengono valutate immediatamente e la Shell Python ne stampa il risultato.

2 è il risultato della valutazione di questa espressione. Si dà il caso che $1 + 1$ sia un'espressione Python valida. Il risultato, naturalmente, è 2.

Proviamone un'altra.

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

Piuttosto semplice, no? Ma potete fare molto di più nella Shell Python. Se doveste mai rimanere bloccati — non riuscite a ricordare un comando, o non riuscite a ricordare gli argomenti appropriati da passare a una certa funzione — potete ottenere aiuto interattivo nella Shell Python. Vi basta digitare `help` e premere INVIO.

```
>>> help
Digitate help() per l'aiuto interattivo, o help(object) per l'aiuto su object.
```

Ci sono due modalità di aiuto. Potete ottenere aiuto su un singolo oggetto, ricavandone semplicemente la stampa della documentazione e ritornando subito al prompt della Shell Python. Potete anche entrare nella *modalità di aiuto*, dove invece di valutare espressioni Python digitate parole chiave e nomi di comandi ottenendo la stampa di tutte le informazioni note su quei comandi.

Per entrare nella modalità di aiuto interattivo, digitate `help()` e premete INVIO.

```
>>> help()
```

Benvenuti in Python 3.0! Questa è la modalità di aiuto in linea.

Se questa è la prima volta che usate Python, dovrete assolutamente dare un'occhiata al tutorial all'indirizzo <http://docs.python.org/tutorial/>.

Digitate il nome di qualsiasi modulo, parola chiave, o argomento per ottenere aiuto su come scrivere programmi Python e usare i moduli Python. Per uscire da questa modalità di aiuto e ritornare all'interprete, vi basta digitare "quit".

Per ottenere una lista di moduli, parole chiave, o argomenti disponibili, digitate "modules", "keywords", o "topics". Ogni modulo è dotato di un breve riepilogo delle sue funzioni; per elencare i moduli i cui riepiloghi contengono una certa parola come "spam", digitate "modules spam".

```
help>
```

Notate come il prompt cambi da >>> a help>. Questo vi ricorda che siete nella modalità di aiuto interattivo. Ora potete inserire qualsiasi parola chiave, comando, nome di modulo, nome di funzione — praticamente qualsiasi cosa comprensibile per Python — e leggere la relativa documentazione.

```
help> print
```

①

Aiuto sulla funzione built-in print nel modulo builtins:

```
print(...)
```

```
print(value, ..., sep=' ', end='\n', file=sys.stdout)
```

Stampa il valore su uno stream, o su sys.stdout per default.

Argomenti con nome opzionali:

file: un oggetto tipo file (stream); usa sys.stdout come valore predefinito.

sep: stringa inserita tra i valori, usa uno spazio come valore predefinito.

end: stringa aggiunta in fondo dopo l'ultimo valore, usa un carattere di ritorno a capo come valore predefinito.

```
help> PapayaWhip
```

②

nessuna documentazione Python trovata per 'PapayaWhip'

```
help> quit
```

③

State uscendo dalla modalità di aiuto e ritornando all'interprete Python.

Se volete chiedere aiuto direttamente su un particolare oggetto da dentro

l'interprete, potete digitare "help(oggetto)". Eseguire "help('stringa')"

ha lo stesso effetto di digitare una particolare stringa al prompt della

modalità di aiuto.

```
>>>
```

④

1. Per ottenere la documentazione sulla funzione print(), vi basta digitare print e premere INVIO. La modalità di aiuto interattivo vi mostrerà qualcosa di simile a una pagina di manuale: il nome della funzione, una breve sinossi, gli argomenti della funzione e i loro valori predefiniti, e così via. Se la documentazione vi sembra oscura, non fatevi prendere dal panico. Imparerete di più su tutti questi concetti nei prossimi capitoli.
2. Naturalmente, la modalità di aiuto interattivo non sa tutto. Se digitate qualcosa che non è un comando, un modulo, una funzione, o un'altra parola chiave predefinita di Python, la modalità di aiuto interattivo si limiterà a scrollare le sue spalle virtuali.

3. Per uscire dalla modalità di aiuto interattivo, digitate `quit` e premete INVIO.
4. Il prompt tornerà a essere `>>>` per segnalarvi che siete usciti dalla modalità di aiuto interattivo e siete ritornati nella Shell Python.

IDLE, la Shell Python grafica, include anche un editor di testo per Python.



0.8. EDITOR E IDE PER PYTHON

IDLE non è l'unica alternativa quando si tratta di scrivere programmi in Python. Sebbene sia utile cominciare imparando semplicemente il linguaggio, molti sviluppatori preferiscono altri editor di testo o ambienti di sviluppo integrati (spesso chiamati IDE, acronimo di Integrated Development Environment). Non ne parlerò qui, ma la comunità Python mantiene una lista di editor per Python che copre un'ampia gamma di piattaforme supportate e di licenze software.

Potreste anche voler controllare la lista di IDE per Python, sebbene pochi supportino già Python 3. Uno di quelli che lo supportano è PyDev, un plug-in per Eclipse che trasforma Eclipse in un IDE Python completo. Sia Eclipse che PyDev sono indipendenti dalla piattaforma e open source.

Sul fronte commerciale, esiste un IDE di ActiveState chiamato Komodo. Le licenze vengono rilasciate al singolo utente, ma gli studenti possono ottenere uno sconto, e sono disponibili versioni gratis per un periodo di prova limitato.

Programmo in Python da nove anni e scrivo i miei programmi Python in GNU Emacs e ne effettuo il debug sulla riga di comando nella Shell Python. Non c'è un modo giusto o sbagliato per sviluppare in Python. Trovate il modo più adatto per voi!

CAPITOLO 1. IL VOSTRO PRIMO PROGRAMMA PYTHON

“ Non nascondere le tue difficoltà dietro un silenzio da santo. Hai un problema? Ottimo. Rallegrati, immergiti e investiga. ”

— Ven. Henepola Gunaratana

1.1. IMMERSIONE!

Come accade per la maggior parte dei libri di programmazione, la convenzione imporrebbe di cominciare con una serie di capitoli noiosi sui fondamentali per poi arrivare gradualmente a costruire qualcosa di utile. Noi ci risparmieremo questa attesa. Qui di seguito troverete subito un programma Python completo e funzionante. Probabilmente ora non ha alcun senso per voi, ma non preoccupatevi perché lo analizzerete riga per riga. Provate comunque a leggerlo, prima di tutto, per vedere se riuscite a ricavarne qualcosa.

```

SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Converte la dimensione di un file in una forma leggibile.

    Argomenti con nome:
    size -- dimensione del file in byte
    a_kilobyte_is_1024_bytes -- se True (default), usa multipli di 1024
                                se False, usa multipli di 1000

    Restituisce: stringa

    ...

    if size < 0:
        raise ValueError('il numero non deve essere negativo')

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('numero troppo grande')

if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))

```

Ora eseguiamo il programma dalla riga di comando. Su Windows avremo qualcosa di simile a questo:

```

c:\home\diveintopython3\esempi> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB

```

Su Mac OS X o Linux avremo qualcosa di simile a questo:

```
you@localhost:~/diveintopython3/esempi$ python3 humansize.py  
1.0 TB  
931.3 GiB
```

Che cos'è appena successo? Avete eseguito il vostro primo programma Python. Avete invocato l'interprete Python dalla riga di comando e gli avete passato il nome dello script che volevate far eseguire all'interprete. Lo script definisce una singola funzione, la funzione `approximate_size()`, che prende una dimensione esatta di un file in byte e ne calcola una versione più “gradevole” (ma approssimata). Lo avete probabilmente visto fare in Esplora Risorse su Windows, o nel Finder di Mac OS X, oppure in Nautilus o Dolphin o Thunar su Linux. Se visualizzate una cartella di documenti come una lista a più colonne, il programma mostrerà una tabella con l'icona di ogni documento, il nome, la dimensione, il tipo, la data dell'ultima modifica, e così via. Se la cartella contiene un file di 1093 byte chiamato `TODO`, il vostro programma di gestione dei file non visualizzerà `TODO 1093 byte`, ma vi mostrerà qualcosa di simile a `TODO 1 KB`. Questo è ciò che fa la funzione `approximate_size()`.

Guardate in fondo al programma e vedrete due chiamate a `print(approximate_size(argomenti))`. Queste sono invocazioni di funzione — prima chiamano la funzione `approximate_size()` passandole un certo numero di argomenti, poi prendono il valore di ritorno e lo passano direttamente alla funzione `print()`. La funzione `print()` è predefinita, perciò non ne vedrete mai la dichiarazione esplicita. Potete semplicemente usarla, sempre e ovunque. (Ci sono molte funzioni predefinite e molte altre funzioni che sono suddivise in *moduli*. Sii paziente, cavalletta.)

Quindi, perché l'esecuzione dello script dalla riga di comando vi restituisce ogni volta gli stessi risultati? Ci arriveremo. Prima di tutto, diamo un'occhiata alla funzione `approximate_size()`.

*
**


1.2. DICHIARARE FUNZIONI

Python vi permette di scrivere funzioni come la maggior parte degli altri linguaggi, ma non utilizza file di intestazione separati come il C++ o sezioni di interface/implementation come il Pascal. Quando avete bisogno di una funzione, vi basta dichiararla, in questo modo:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

La parola chiave `def` inizia la dichiarazione di funzione, seguita dal nome della funzione, seguito dagli argomenti della funzione racchiusi tra parentesi. Gli argomenti sono separati da virgole.

Notate anche che la funzione non definisce un tipo di dato di ritorno. Le funzioni Python non specificano il tipo di dato del loro valore di ritorno, non specificano nemmeno se restituiscono o no un valore. (In effetti, ogni funzione Python restituisce un valore: se la funzione esegue un'istruzione `return`, restituirà quel valore, altrimenti restituirà `None`, il valore nullo di Python.)

 In alcuni linguaggi, le funzioni (che restituiscono un valore) cominciano con `function` e le procedure (che non restituiscono un valore) cominciano con `sub`. Non ci sono procedure in Python. Ci sono solo funzioni, tutte le funzioni restituiscono un valore (anche se è `None`) e tutte le funzioni cominciano con `def`.

*Quando avete
bisogno di
una funzione,
vi basta
dichiararla.*

La funzione `approximate_size()` accetta due argomenti — `size` e `a_kilobyte_is_1024_bytes` — ma nessun argomento dichiara il proprio tipo. In Python, le variabili non sono mai esplicitamente tipate. L'interprete Python capisce da solo qual è il tipo di una variabile e ne tiene traccia internamente.

☞ In Java e in altri linguaggi staticamente tipati, dovete dichiarare il tipo del valore di ritorno e di ogni argomento della funzione. In Python, i tipi non vengono mai dichiarati. Sulla base del valore che viene assegnato, Python tiene traccia del tipo di dato internamente.

1.2.1. ARGOMENTI OPZIONALI E CON NOME

Python consente agli argomenti di funzione di avere valori predefiniti, in modo che, se una funzione viene chiamata senza un argomento, quell'argomento venga impostato al suo valore predefinito. In più, gli argomenti possono essere specificati in qualsiasi ordine usando argomenti con nome.

Diamo un'altra occhiata a quella dichiarazione della funzione `approximate_size()`:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

Il secondo argomento, `a_kilobyte_is_1024_bytes`, specifica un valore predefinito pari a `True`. Questo significa che l'argomento è *opzionale*: potete invocare la funzione senza di esso e Python opererà come se l'aveste invocata passando `True` come secondo parametro.

Ora guardate in fondo allo script:

```
if __name__ == '__main__':  
    print(approximate_size(1000000000000, False)) ①  
    print(approximate_size(1000000000000))        ②
```

1. Questa riga invoca la funzione `approximate_size()` con due argomenti. All'interno della funzione `approximate_size()`, `a_kilobyte_is_1024_bytes` varrà `False`, dato che avete esplicitamente passato `False` come secondo argomento.
2. Questa riga invoca la funzione `approximate_size()` con un solo argomento. Ma questo va ancora bene, perché il secondo argomento è opzionale! Dato che il chiamante non lo specifica, il secondo argomento assumerà il proprio valore predefinito `True`, come stabilito nella dichiarazione di funzione.

I valori degli argomenti di una funzione possono anche essere passati per nome.

```

>>> from humansize import approximate_size

>>> approximate_size(4000, a_kilobyte_is_1024_bytes=False) ①
'4.0 KB'

>>> approximate_size(size=4000, a_kilobyte_is_1024_bytes=False) ②
'4.0 KB'

>>> approximate_size(a_kilobyte_is_1024_bytes=False, size=4000) ③
'4.0 KB'

>>> approximate_size(a_kilobyte_is_1024_bytes=False, 4000) ④
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg

>>> approximate_size(size=4000, False) ⑤
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg

```

1. Questa è un'invocazione della funzione `approximate_size()` con `4000` come valore per il primo argomento (`size`) e `False` come valore per il secondo argomento chiamato `a_kilobyte_is_1024_bytes`. (Si dà il caso che questo sia effettivamente il secondo argomento, ma la cosa non ha importanza, come vedrete fra un minuto.)
2. Questa è un'invocazione della funzione `approximate_size()` con `4000` come valore per l'argomento chiamato `size` e `False` come valore per l'argomento chiamato `a_kilobyte_is_1024_bytes`. (Si dà il caso che questi argomenti con nome siano nello stesso ordine in cui sono elencati gli argomenti nella dichiarazione di funzione, ma anche questo non ha importanza.)
3. Questa è un'invocazione della funzione `approximate_size()` con `False` come valore per l'argomento chiamato `a_kilobyte_is_1024_bytes` e `4000` come valore per l'argomento chiamato `size`. (Vedete? Vi avevo detto che l'ordine non ha importanza.)
4. Questa invocazione fallisce, perché passate un argomento con nome seguito da un argomento (di tipo posizionale) senza nome, e questo non funziona mai. Leggendo la lista degli argomenti da sinistra a destra, una volta che utilizzate un singolo argomento con nome anche il resto degli argomenti deve essere dello stesso tipo.
5. Anche questa invocazione fallisce, per lo stesso motivo della precedente. Vi sorprende? Dopo tutto, avete passato `4000` come valore per l'argomento chiamato `size`, poi “ovviamente” intendevate passare quel valore `False` all'argomento `a_kilobyte_is_1024_bytes`. Ma Python non funziona in questo modo. Appena utilizzate un argomento con nome, anche tutti gli argomenti alla destra di quell'argomento devono essere argomenti con nome.



1.3. SCRIVERE CODICE LEGGIBILE

Non vi annoierò con una lunga predica sull'importanza di documentare il vostro codice. Vi basti sapere che il codice viene scritto una volta sola ma letto molte volte, e che i lettori più importanti del vostro codice siete voi stessi, sei mesi dopo averlo scritto (cioè dopo che vi siete dimenticati tutto ma avete bisogno di correggere qualcosa). Python facilita la scrittura di codice leggibile, perciò avvantaggiatevene. Fra sei mesi mi ringrazierete.

1.3.1. STRINGHE DI DOCUMENTAZIONE

Potete documentare una funzione Python aggiungendole una stringa di documentazione (chiamata docstring per brevità). Nel nostro programma, la funzione `approximate_size()` è stata dotata di una docstring:


```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Converte la dimensione di un file in una forma leggibile.

    Argomenti con nome:
    size -- dimensione del file in byte
    a_kilobyte_is_1024_bytes -- se True (default), usa multipli di 1024
                                se False, usa multipli di 1000

    Restituisce: stringa


    ...
```

Le triple virgolette servono per rappresentare una stringa su più righe. Ogni cosa tra le virgolette di inizio e di fine è parte di una singola stringa, inclusi i ritorni a capo, gli spazi bianchi all'inizio di una riga e altri caratteri di virgolette. Le triple virgolette si possono usare dovunque, ma le vedrete sfruttate soprattutto nella definizione di una `docstring`.

 Le triple virgolette sono anche un modo facile per definire una stringa che contiene sia apici che virgolette, come `qq/.../` in Perl 5.

*Ogni
funzione
merita una
docstring
decente.*

Tutto quello che si trova tra le triple virgolette rappresenta la `docstring` della funzione, che documenta ciò che la funzione fa. Una `docstring`, se esiste, deve essere la prima cosa definita in una funzione (cioè, nella riga subito dopo la dichiarazione di funzione). Non avete tecnicamente bisogno di dotare la vostra funzione di una `docstring`, ma dovrete sempre farlo. So che lo avete sentito in ogni corso di programmazione che avete seguito, ma Python vi dà un incentivo aggiuntivo: la `docstring` è disponibile a tempo di esecuzione sotto forma di attributo della funzione.

 Molti IDE per Python usano le `docstring` per fornire documentazione sensibile al contesto, in modo che quando scrivete il nome di una funzione la sua `docstring` appaia sotto forma di suggerimento. Questa caratteristica può essere incredibilmente utile, ma lo è solamente tanto quanto le `docstring` che scrivete.



1.4. IL PERCORSO DI RICERCA DI import

Prima di proseguire, voglio menzionare brevemente il percorso di ricerca delle librerie. Python guarda in diversi posti quando provate a importare un modulo. Nello specifico, guarda in tutte le directory definite in `sys.path`. Questo attributo è semplicemente una lista e potete facilmente vederlo o modificarlo con i metodi standard delle liste. (Imparerete di più sulle liste nel capitolo sui tipi di dato nativi.)

```
>>> import sys                                ①
>>> sys.path                                  ②
['',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']

>>> sys                                        ③
<module 'sys' (built-in)>

>>> sys.path.insert(0, '/home/mark/diveintopython3/esempi') ④
>>> sys.path                                  ⑤
['/home/mark/diveintopython3/esempi',
 '',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
```

1. Importare il modulo `sys` rende disponibili tutte le sue funzioni e i suoi attributi.
2. `sys.path` è una lista di nomi di directory che costituiscono il percorso di ricerca corrente. (Il vostro apparirà diverso, a seconda del vostro sistema operativo, della versione di Python che state eseguendo e di dove è stata originariamente installata.) Python cercherà attraverso queste directory (in questo ordine) un file con estensione `.py` il cui nome corrisponda a quello che state tentando di importare.

3. In realtà, ho mentito; la verità è più complicata di così, perché non tutti i moduli sono memorizzati come file `.py`. Alcuni sono moduli *built-in*, integrati direttamente all'interno dell'interprete Python. I moduli built-in si comportano esattamente come gli altri moduli, ma il loro codice sorgente Python non è disponibile, perché non sono scritti in Python! (Come lo stesso interprete del linguaggio, questi moduli sono scritti in C.)
4. Potete aggiungere a tempo di esecuzione una nuova directory al percorso di ricerca di Python aggiungendo il nome della directory a `sys.path`; successivamente, Python guarderà anche in quella directory ogni volta che provate a importare un modulo. L'effetto dura fino a quando l'interprete Python è in esecuzione.
5. Usando `sys.path.insert(0, nuovo_percorso)`, avete inserito una nuova directory come primo elemento della lista `sys.path`, quindi all'inizio del percorso di ricerca di Python. Questo è quasi sempre quello che desiderate. In caso di un conflitto di nomi (per esempio, nel caso l'interprete Python includa la versione 2 di una particolare libreria ma voi vogliate usarne la versione 3), questo vi assicura che saranno i vostri moduli a essere trovati e utilizzati invece dei moduli distribuiti insieme all'interprete Python.



1.5. OGNI COSA È UN OGGETTO

Nel caso vi fosse sfuggito, ho appena detto che le funzioni Python sono dotate di attributi e che questi attributi sono disponibili a tempo di esecuzione. Una funzione, come ogni altra cosa in Python, è un oggetto.

Lanciate la shell interattiva di Python e seguitemi attraverso questa serie di istruzioni:

```
>>> import humansize ①
```

```
>>> print(humansize.approximate_size(4096, True)) ②
```

```
4.0 KiB
```

```
>>> print(humansize.approximate_size.__doc__) ③
```

```
Converte la dimensione di un file in una forma leggibile.
```

Argomenti con nome:

`size` -- dimensione del file in byte

`a_kilobyte_is_1024_bytes` -- se `True` (default), usa multipli di 1024
se `False`, usa multipli di 1000

Restituisce: stringa

1. La prima riga importa il programma `humansize` come un modulo — un pezzo di codice che potete usare interattivamente o da un programma Python più grande. Una volta importato un modulo, potete fare riferimento a ogni funzione, classe, o attributo pubblico che appartiene al modulo. I moduli possono sfruttare questa loro caratteristica per accedere alle funzionalità di altri moduli, e voi potete farlo anche nella shell interattiva di Python. Questo è un concetto importante, e lo vedrete ribadito più volte nel seguito di questo libro.
2. Quando volete usare funzioni definite in moduli importati, dovete includere il nome del modulo. Quindi non potete semplicemente dire `approximate_size`, ma dovete scrivere `humansize.approximate_size`. Se avete usato le classi in Java, la sintassi dovrebbe risultarvi vagamente familiare.
3. Invece di invocare la funzione, avete richiesto uno dei suoi attributi, `__doc__`.

☞ `import` in Python è come `require` in Perl. Una volta che avete importato un modulo Python tramite `import`, potete accedere alle sue funzioni con la sintassi `modulo.funzione`; una volta che avete richiesto un modulo Perl tramite `require`, potete accedere alle sue funzioni con la sintassi `modulo::funzione`.

1.5.1. CHE COS'È UN OGGETTO?

Ogni cosa in Python è un oggetto, e ogni cosa può possedere attributi e metodi. Tutte le funzioni hanno un attributo built-in `__doc__`, che restituisce la `docstring` definita nel codice sorgente della funzione. Il modulo `sys` è un oggetto che ha (tra le altre cose) un attributo chiamato `path`. E così via.

Tuttavia, questo non risponde alla domanda fondamentale: che cos'è un oggetto? Linguaggi di programmazione differenti definiscono “oggetto” in modi differenti. In alcuni, significa che *tutti* gli oggetti *devono* avere attributi e metodi; in altri, significa che tutti gli oggetti sono estendibili. In Python, la definizione è più lasca. Alcuni oggetti non hanno attributi né metodi, *ma potrebbero averli*. Non tutti gli oggetti sono estendibili. Ma ogni cosa è un oggetto nel senso che può essere assegnata a una variabile o passata come argomento a una funzione.

Potreste aver udito il termine “oggetto di prima classe” in altri contesti legati alla programmazione. In Python, le funzioni sono *oggetti di prima classe*. Potete passare una funzione come argomento a un'altra funzione. I moduli sono *oggetti di prima classe*. Potete passare un intero modulo come argomento a una funzione. Le classi sono oggetti di prima classe e anche le singole istanze di una classe sono oggetti di prima classe.

Questo concetto è importante, quindi lo ripeterò nel caso vi fosse sfuggito le prime volte: *ogni cosa in Python è un oggetto*. Le stringhe sono oggetti. Le liste sono oggetti. Le funzioni sono oggetti. Le classi sono oggetti. Le istanze di una classe sono oggetti. Persino i moduli sono oggetti.



1.6. INDENTARE IL CODICE

Le funzioni Python non cominciano esplicitamente con `begin` né terminano con `end`, e non ci sono parentesi graffe a indicare dove il codice della funzione inizia e finisce. I due punti (`:`) e l'indentazione del codice sono gli unici delimitatori.

```

def approximate_size(size, a_kilobyte_is_1024_bytes=True): ①
    if size < 0: ②
        raise ValueError('il numero non deve essere negativo') ③
    ④
    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]: ⑤
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('numero troppo grande')

```

1. I blocchi di codice sono definiti dalla loro indentazione. Per “blocco di codice” intendo funzioni, istruzioni `if`, cicli `for`, cicli `while`, e così via. La presenza di un’indentazione comincia un blocco e la sua assenza lo termina. Non ci sono parentesi graffe, parentesi quadre, o parole chiave. Questo vuol dire che gli spazi bianchi sono significativi e devono essere consistenti. In questo esempio, il codice della funzione è indentato di quattro spazi. Non è necessario che siano quattro spazi, c’è solo bisogno che l’indentazione sia consistente. La prima riga che non è indentata indica la fine della funzione.
2. In Python, un’istruzione `if` è seguita da un blocco di codice. Se l’espressione di `if` viene valutata come vera, il blocco indentato viene eseguito, altrimenti si ricade nel blocco di `else` (se esiste). Notate la mancanza di parentesi attorno all’espressione.
3. Questa riga è dentro il blocco di codice dell’`if`. L’istruzione di `raise` solleverà un’eccezione (di tipo `ValueError`), ma solo se `size < 0`.
4. Questa *non* è la fine della funzione. Le righe completamente vuote non vengono considerate. Possono rendere il codice più leggibile, ma non vengono usate per delimitare i blocchi. La funzione continua nella riga seguente.
5. Anche il ciclo `for` segnala l’inizio di un blocco di codice. I blocchi di codice possono contenere più righe, fino a quando esse sono indentate della stessa quantità. Questo ciclo `for` contiene tre righe di codice. Non c’è nessun’altra sintassi speciale per blocchi di codice su più righe. Indentate e andate avanti con la vostra vita.

Dopo alcune proteste iniziali e diverse analogie maliziose al Fortran, vi riappacificherete con questa caratteristica di Python e comincerete a vederne i lati positivi. Uno dei maggiori benefici è che tutti i programmi Python appaiono simili, in quanto l’indentazione è un requisito del linguaggio e non una questione di stile. Questo rende più facile leggere e comprendere il codice Python scritto da altre persone.

☞ Python usa i ritorni a capo per separare le istruzioni e i due punti e l'indentazione per separare i blocchi di codice. C++ e Java usano i punti e virgola per separare le istruzioni e le parentesi graffe per separare i blocchi di codice.

*
**

1.7. ECCEZIONI

Le eccezioni sono dappertutto in Python. Virtualmente ogni modulo nella libreria standard le usa e lo stesso interprete Python le solleverà in molte circostanze differenti. Le vedrete ripetutamente nel corso di questo libro.


Che cos'è un'eccezione? Di solito è un errore, un'indicazione che qualcosa è andata storta. (Non tutte le eccezioni sono errori, ma non preoccupatevi di questo per ora.) Alcuni linguaggi di programmazione incoraggiano l'uso di codici di errore di ritorno, che vengono *controllati*. Python incoraggia l'uso delle eccezioni, che vengono *gestite*.

Quando avviene un errore nella Shell Python, vengono stampati alcuni dettagli sull'eccezione e su come è avvenuta, e questo è tutto. Questa viene chiamata un'eccezione *non gestita*. Quando l'eccezione è stata sollevata, non c'era alcuna istruzione per notarla esplicitamente e affrontarla, quindi è gorgogliata in superficie fino al livello della Shell Python, che sputa fuori alcune informazioni di debug e considera finito il proprio lavoro. Nella shell questo non è un grande problema, ma se accadesse mentre il vostro programma Python è in esecuzione, l'intero programma si bloccherebbe nel caso l'eccezione non venisse gestita. Magari questo è il comportamento che desiderate, magari non lo è.

☞ A differenza di Java, le funzioni Python non dichiarano quali eccezioni potrebbero sollevare. Sta a voi determinare quali possibili eccezioni avete bisogno di catturare.

Un'eccezione non deve necessariamente risultare in un completo blocco del programma, comunque. Le eccezioni possono essere *gestite*. A volte un'eccezione compare perché in realtà il vostro codice ha un bug


(come il tentativo di accedere a una variabile che non esiste), ma altre volte un'eccezione è qualcosa che potete anticipare. Se state aprendo un file, quel file potrebbe non esistere. Se state importando un modulo, quel modulo potrebbe non essere installato. Se state aprendo una connessione a un database, quel database potrebbe non essere disponibile, oppure potreste non avere le credenziali di sicurezza corrette per accedere. Se sapete che una riga di codice potrebbe sollevare un'eccezione, dovrete gestire l'eccezione usando un blocco `try...except`.

 Python usa blocchi `try...except` per gestire le eccezioni e l'istruzione `raise` per generarle. Java e C++ usano blocchi `try...catch` per gestire le eccezioni e l'istruzione `throw` per generarle.

La funzione `approximate_size()` solleva eccezioni in due casi differenti: se la variabile `size` passata contiene una dimensione più grande di quella che la funzione è progettata per gestire, o se contiene una dimensione inferiore a zero.

```
if size < 0:
    raise ValueError('il numero non deve essere negativo')
```

La sintassi per sollevare un'eccezione è abbastanza semplice. Usate l'istruzione `raise`, seguita dal nome dell'eccezione e da una stringa contenente un messaggio per scopi di debug. La sintassi ricorda quella dell'invocazione di una funzione. (In realtà, le eccezioni sono implementate come classi e questa istruzione `raise` crea effettivamente un'istanza della classe `ValueError` passando la stringa 'il numero non deve essere negativo' al suo metodo di inizializzazione. Ma stiamo mettendo il carro avanti ai buoi!)

 Non avete bisogno di gestire un'eccezione nella funzione che la solleva. Se una funzione non gestisce un'eccezione, l'eccezione viene passata alla funzione chiamante, poi alla funzione che ha chiamato quella funzione, e così via “fino alla cima dello stack di esecuzione”. Se l'eccezione non viene mai gestita, il vostro programma si bloccherà, Python stamperà una “traccia dello stack” sul canale di errore e le cose finiranno lì. Ancora una volta, forse questo è il comportamento che desiderate; dipende da ciò che fa il vostro programma.

1.7.1. CATTURARE GLI ERRORI DI IMPORTAZIONE

Una delle eccezioni predefinite di Python è `ImportError`, che viene sollevata quando cercate di importare un modulo e l'operazione fallisce. Questo può succedere per una varietà di ragioni, ma il caso più semplice è quello in cui il modulo non esiste nel vostro percorso di ricerca per le importazioni. Potete usare questa eccezione per includere funzionalità opzionali nel vostro programma. Per esempio, la libreria `chardet` fornisce il riconoscimento automatico delle codifiche di carattere. Forse il vostro programma desidera usare questa libreria se esiste, ma continuare normalmente se l'utente non l'ha installata. Potete fare questo con un blocco `try..except`.

```
try:
    import chardet
except ImportError:
    chardet = None
```

Più tardi, potete controllare la presenza del modulo `chardet` con una semplice istruzione `if`:

```
if chardet:
    # fai qualcosa
else:
    # continua comunque
```

L'eccezione `ImportError` viene comunemente usata anche quando due moduli implementano una API comune, ma uno è più desiderabile dell'altro perché magari è più veloce o usa meno memoria. Potete provare a importare un modulo ma ripiegare su un modulo differente se la prima importazione fallisce. Per esempio, il capitolo su XML parla di due moduli che implementano una API comune chiamata `ElementTree`. Il primo, `lxml`, è un modulo di terze parti che dovete scaricare e installare da voi. Il secondo, `xml.etree.ElementTree`, è più lento ma è parte della libreria standard di Python 3.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

Alla fine di questo blocco `try...except` avrete importato un *qualche* modulo e l'avrete chiamato `etree`. Dato che entrambi i moduli implementano una API comune, il resto del vostro programma non deve controllare ogni volta quale modulo è stato importato. E dato che il modulo che è stato importato viene sempre chiamato `etree`, non c'è bisogno di spargere istruzioni `if` ovunque nel resto del vostro programma per chiamare moduli con nomi differenti.

*
**

1.8. VARIABILI NON LEGATE

Date un'altra occhiata a questa riga di codice nella funzione `approximate_size()`:

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

Non dichiarate mai la variabile `multiple`, ma le assegnate semplicemente un valore. Questo va bene, perché Python vi permette di farlo. Quello che Python *non* vi permette è fare riferimento a una variabile a cui non è mai stato assegnato un valore. Il tentativo di fare questo solleverà un'eccezione di tipo `NameError`.

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined

>>> x = 1
>>> x
1
```

Un giorno ringrazierete Python per questo.

*
**

1.9. OGNI COSA È SENSIBILE ALLE MAIUSCOLE

Tutti i nomi in Python sono sensibili alle maiuscole: nomi di variabile, nomi di funzione, nomi di classe, nomi di modulo, nomi di eccezione. Se potete ottenerlo, impostarlo, invocarlo, costruirlo, importarlo, o sollevarlo, allora è sensibile alle maiuscole.

```
>>> an_integer = 1
>>> an_integer
1
>>> AN_INTEGER
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'AN_INTEGER' is not defined
>>> An_Integer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'An_Integer' is not defined
>>> an_inteGer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'an_inteGer' is not defined
```

E così via.

*
**


1.10. ESEGUIRE GLI SCRIPT

I moduli Python sono oggetti e hanno diversi attributi utili. Potete sfruttare questa caratteristica per collaudare facilmente i vostri moduli mentre li scrivete, includendo uno speciale blocco di codice che viene eseguito quando lanciate il file Python dalla riga di comando. Considerate le ultime righe di `humansize.py`:

```
if __name__ == '__main__':
```

```
    print(approximate_size(1000000000000, False))
```

```
    print(approximate_size(1000000000000))
```

 Come il C, Python usa `==` per il confronto e `=` per l'assegnamento. A differenza del C, Python non supporta l'assegnamento in linea, quindi non c'è alcuna possibilità di assegnare accidentalmente il valore che pensavate di stare confrontando.

Che cosa rende speciale questa istruzione `if`? Ebbene, i moduli sono oggetti, e tutti i moduli hanno l'attributo built-in `__name__`. Il valore dell'attributo `__name__` di un modulo dipende da come state usando il modulo. Se importate un modulo, allora `__name__` sarà il nome del file del modulo, senza il percorso e l'estensione.

```
>>> import humansize
>>> humansize.__name__
'humansize'
```

*Ogni cosa in
Python è un
oggetto.*

Ma potete anche eseguire direttamente il modulo come un programma a sé, nel qual caso `__name__` sarà lo speciale valore predefinito `__main__`. Nel nostro esempio, Python valuterà l'istruzione `if`, troverà un'espressione vera ed eseguirà il blocco di codice corrispondente. In questo caso, per stampare due valori.

```
c:\home\diveintopython3> c:\python31\python.exe humansize.py
```

```
1.0 TB
```

```
931.3 GiB
```

E questo è il vostro primo programma Python!



1.11. LETTURE DI APPROFONDIMENTO

- PEP 257: Convenzioni per le docstring spiega cosa distingue una buona docstring da una docstring eccellente.
- Tutorial Python: stringhe di documentazione si occupa a sua volta dell'argomento.
- PEP 8: Guida allo stile per il codice Python descrive il buon stile di indentazione.
- Guida di riferimento a Python spiega cosa significa dire che ogni cosa in Python è un oggetto, perché alcune persone sono pedanti e adorano discutere questo tipo di cose in lungo e in largo.

CAPITOLO 2. TIPI DI DATO NATIVI

“ Lo stupore è la base di ogni filosofia, la ricerca ne è il procedimento, l'ignoranza la fine. ”
— Michel de Montaigne

2.1. IMMERSIONE!

Disinteressatevi del vostro primo programma Python per un minuto, e parliamo di tipi di dato. In Python, ogni valore ha un tipo di dato, ma non è necessario dichiarare il tipo di dato delle variabili. Come mai? In base all'assegnamento originale di ogni variabile, Python capisce di quale tipo è e ne tiene traccia internamente.

Python ha molti tipi di dato nativi. Questi sono quelli più importanti:

1. **Booleani**: sono solamente True oppure False.
2. **Numeri**: possono essere interi (1 e 2), reali (1.1 e 1.2), frazioni (1/2 e 2/3), o persino numeri complessi.
3. **Stringhe**: sono sequenze di caratteri Unicode, per esempio un documento HTML.
4. **Byte e byte array**: per esempio un file JPEG contenente un'immagine.
5. **Liste**: sono sequenze ordinate di valori.
6. **Tuple**: sono sequenze di valori ordinate e immutabili.
7. **Insiemi**: sono gruppi non ordinati di valori.
8. **Dizionari**: sono gruppi non ordinati di coppie chiave-valore.

Naturalmente, esistono molti più tipi di questi. Ogni cosa è un oggetto in Python, quindi ci sono anche tipi come *modulo*, *funzione*, *classe*, *metodo*, *file* e persino *codice compilato*. Ne avete già visti alcuni: i moduli hanno un nome, le funzioni sono dotate di una docstring, &c. Imparerete cosa sono le classi nel capitolo Classi & iteratori e cosa sono i file nel capitolo File.

Stringhe e byte sono abbastanza importanti — e abbastanza complicati — da essere trattati in un capitolo a parte. Diamo un'occhiata agli altri per primi.



2.2. BOOLEANI

I dati di tipo booleano sono veri o falsi. Python ha due costanti, chiamate ingegnosamente `True` e `False`, che possono essere usate per assegnare direttamente valori di tipo booleano. Anche le espressioni possono essere valutate come valori booleani. In certi posti (come le istruzioni `if`), Python si aspetta un'espressione da valutare come un valore booleano. Questi posti sono chiamati *contesti logici*. Potete virtualmente usare qualsiasi espressione in un contesto logico, e Python proverà a determinarne il valore di verità. Tipi di dato differenti hanno regole differenti per decidere se un loro valore è vero o falso in un contesto logico. (Questa frase avrà più senso una volta che avrete visto alcuni esempi concreti più avanti in questo capitolo.)

Per esempio, considerate questo estratto da

`humansize.py`:

```
if size < 0:
    raise ValueError('il numero non deve essere negativo')
```

`size` è un intero, `0` è un intero, e `<` è un operatore numerico. Il risultato dell'espressione `size < 0` è sempre un booleano. Potete verificarlo voi stessi nella shell interattiva di Python:

*Potete usare
virtualmente
qualsiasi
espressione in
un contesto
logico.*

```
>>> size = 1
>>> size < 0
False
>>> size = 0
>>> size < 0
False
>>> size = -1
>>> size < 0
True
```

A causa di alcune questioni lasciate in eredità da Python 2, i booleani possono essere trattati come numeri. True vale 1 e False vale 0.

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Ow, ow, ow! Non fatelo. Dimenticatevi persino che io lo abbia menzionato.

*
**

2.3. NUMERI

I numeri sono meravigliosi. Ce ne sono così tanti tra cui scegliere. Python supporta sia gli interi che i numeri in virgola mobile. Non c'è alcuna dichiarazione di tipo per distinguerli, perché Python è in grado di farlo grazie alla presenza o meno del punto decimale.

```
>>> type(1)                ①
<class 'int'>

>>> isinstance(1, int)     ②
True

>>> 1 + 1                  ③
2

>>> 1 + 1.0                ④
2.0

>>> type(2.0)
<class 'float'>
```

1. Potete usare la funzione `type()` per controllare il tipo di qualsiasi valore o variabile. Come vi sareste aspettati, 1 è un `int`.
2. Similmente, potete usare la funzione `isinstance()` per controllare se un valore o una variabile è di un certo tipo.
3. Aggiungere un `int` a un `int` dà come risultato un `int`.
4. Aggiungere un `int` a un `float` dà come risultato un `float`. Python converte implicitamente l'`int` in un `float` per eseguire l'addizione, poi restituisce un `float` come risultato.

2.3.1. CONVERTIRE INTERI IN REALI E VICEVERSA

Come avete appena visto, alcuni operatori (come l'addizione) convertono implicitamente gli interi in numeri in virgola mobile su necessità. Potete anche convertirli esplicitamente voi stessi.

```

>>> float(2)           ①
2.0

>>> int(2.0)           ②
2

>>> int(2.5)           ③
2


>>> int(-2.5)          ④
-2

>>> 1.12345678901234567890 ⑤
1.1234567890123457

>>> type(1000000000000000) ⑥
<class 'int'>

```

1. Potete convertire esplicitamente un `int` in un `float` chiamando la funzione `float()`.
2. Naturalmente, potete anche convertire esplicitamente un `float` in un `int` chiamando `int()`.
3. La funzione `int()` esegue un troncamento, non un arrotondamento.
4. La funzione `int()` tronca i numeri negativi verso lo 0. È una vera funzione di troncamento, non una funzione di parte intera.
5. I numeri in virgola mobile sono accurati fino a 15 decimali.
6. Gli interi possono essere arbitrariamente grandi.

 Python 2 era dotato di tipi separati per `int` e `long`. Il tipo di dato `int` era limitato da `sys.maxint`, che variava a seconda della piattaforma ma di solito valeva $2^{32}-1$. Python 3 possiede un solo tipo per i numeri interi, che si comporta quasi sempre come il vecchio tipo `long` di Python 2. Si veda la [PEP 237](#) per i dettagli.

2.3.2. OPERAZIONI COMUNI SUI NUMERI

Potete fare ogni tipo di operazione con i numeri.

```

>>> 11 / 2      ①
5.5
>>> 11 // 2     ②
5
>>> -11 // 2    ③
-6
>>> 11.0 // 2   ④
5.0
>>> 11 ** 2     ⑤
121
>>> 11 % 2      ⑥
1

```

1. L'operatore `/` esegue la divisione in virgola mobile. Restituisce un `float` anche quando il numeratore e il denominatore sono entrambi numeri di tipo `int`.
2. L'operatore `//` esegue un tipo strambo di divisione tra interi. Quando il risultato è positivo, potete immaginarla come un troncamento (non un arrotondamento) a 0 decimali, ma fate attenzione.
3. Quando eseguite la divisione intera tra numeri negativi, l'operatore `//` arrotonda “per eccesso” verso l'intero più vicino. Matematicamente parlando, l'arrotondamento è “per difetto” dato che `-6` è inferiore a `-5`, ma se vi aspettate un troncamento a `-5` potreste essere indotti in errore.
4. L'operatore `//` non restituisce sempre un intero. Se il numeratore o il denominatore sono di tipo `float`, effettuerà un arrotondamento all'intero più vicino, ma l'effettivo valore di ritorno sarà un `float`.
5. L'operatore `**` significa “elevato alla potenza di”. 11^2 vale 121.
6. L'operatore `%` dà il resto dopo aver effettuato una divisione di tipo intero. 11 diviso per 2 vale 5 con il resto di 1, quindi il risultato in questo caso è 1.



In Python 2, l'operatore `/` normalmente effettuava una divisione di tipo intero, ma avreste potuto obbligarlo a comportarsi come la divisione in virgola mobile includendo una speciale direttiva nel vostro codice. In Python 3, l'operatore `/` effettua sempre una divisione in virgola mobile. Si veda la [PEP 238](#) per i dettagli.

2.3.3. FRAZIONI

Python non si limita agli interi e ai numeri in virgola mobile. Può anche fare tutta quella matematica elaborata che avete imparato alle superiori e poi prontamente dimenticato.

```
>>> import fractions ①
>>> x = fractions.Fraction(1, 3) ②
>>> x
Fraction(1, 3)
>>> x * 2 ③
Fraction(2, 3)
>>> fractions.Fraction(6, 4) ④
Fraction(3, 2)
>>> fractions.Fraction(0, 0) ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fractions.py", line 96, in __new__
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(0, 0)
```

1. Per cominciare a usare le frazioni, importate il modulo `fractions`.
2. Per definire una frazione, create un oggetto `Fraction` e passategli il numeratore e il denominatore.
3. Potete eseguire tutte le comuni operazioni matematiche sulle frazioni. Le operazioni restituiscono un nuovo oggetto `Fraction`. $2 * (1/3) = (2/3)$
4. L'oggetto `Fraction` semplificherà le frazioni automaticamente. $(6/4) = (3/2)$
5. Python ha il buon senso di non creare frazioni con denominatore uguale a zero.

2.3.4. TRIGONOMETRIA

Potete anche operare con la trigonometria di base in Python.


```
>>> import math
>>> math.pi ①
3.1415926535897931
>>> math.sin(math.pi / 2) ②
1.0
>>> math.tan(math.pi / 4) ③
0.9999999999999999
```

1. Il modulo `math` include una costante per π , il rapporto tra la circonferenza di un cerchio e il suo diametro.
2. Il modulo `math` contiene tutte le funzioni trigonometriche di base, comprese `sin()`, `cos()`, `tan()` e varianti come `asin()`.
3. Notate, comunque, che la precisione numerica di Python non è infinita. `tan(π / 4)` dovrebbe restituire `1.0`, non `0.9999999999999999`.

2.3.5. NUMERI IN UN CONTESTO LOGICO

Potete usare i numeri in un contesto logico come quello di un'istruzione `if`. I valori uguali a zero sono falsi e i valori diversi da zero sono veri.

*I valori
uguali a zero
sono falsi e i
valori diversi*

```

>>> def is_it_true(anything): ①
...     if anything:
...         print('sì, è vero')
...     else:
...         print('no, è falso')
...
>>> is_it_true(1) ②
sì, è vero
>>> is_it_true(-1)
sì, è vero
>>> is_it_true(0)
no, è falso
>>> is_it_true(0.1) ③
sì, è vero
>>> is_it_true(0.0)
no, è falso
>>> import fractions
>>> is_it_true(fractions.Fraction(1, 2)) ④
sì, è vero
>>> is_it_true(fractions.Fraction(0, 1))
no, è falso

```

*da zero sono
veri.*

1. Sapevate di poter definire le vostre funzioni nella shell interattiva di Python? Vi basta premere INVIO alla fine di ogni riga, e INVIO su una riga vuota per concludere la definizione.
2. In un contesto logico, gli interi diversi da zero sono veri e 0 è falso.
3. I numeri in virgola mobile diversi da zero sono veri e 0.0 è falso. Fate attenzione! Se c'è anche il più piccolo errore di arrotondamento (non impossibile, come avete visto nella sezione precedente) allora Python opererà su 0.00000000000001 invece di 0 e restituirà True.
4. Anche le frazioni possono essere usate in un contesto logico. Fraction(0, n) è falso per tutti i valori di n. Tutte le altre frazioni sono vere.



2.4. LISTE

Le liste sono il mulo da soma dei tipi di dato Python. Quando dico “lista” potreste pensare a un “array la cui dimensione devo dichiarare in anticipo e che può solo contenere elementi di uno stesso tipo, &c”. Non pensatelo. Le liste sono molto più fighe di così.

☞ Una lista in Python è come un array in Perl 5. In Perl 5, le variabili che memorizzano un array cominciano sempre con il carattere @; in Python, le variabili possono essere chiamate in qualsiasi modo, e Python tiene traccia del tipo di dato internamente.

☞ Una lista in Python è molto più di un array in Java (anche se può essere usata come tale, se questo è tutto quello che volete dalla vita). Un’analogia migliore sarebbe quella con la classe `ArrayList`, che può contenere oggetti di tipo arbitrario e può espandersi dinamicamente man mano che nuovi elementi vengono aggiunti.

2.4.1. CREARE UNA LISTA

Creare una lista è facile: usate le parentesi quadre per racchiudere un elenco di valori separati da virgole.

```
>>> a_list = ['a', 'b', 'mpilgrim', 'z', 'esempio'] ①
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'esempio']
>>> a_list[0] ②
'a'
>>> a_list[4] ③
'esempio'
>>> a_list[-1] ④
'esempio'
>>> a_list[-3] ⑤
'mpilgrim'
```

1. Per prima cosa, definite una lista di cinque elementi. Notate che essi mantengono il loro ordine originale. Questo non è un caso, poiché una lista è un insieme ordinato di elementi.
2. Una lista può essere usata come un array il cui indice parte da zero. Il primo elemento di una lista non vuota è sempre `a_list[0]`.
3. L'ultimo elemento di questa lista di cinque elementi è `a_list[4]`, perché l'indice delle liste parte sempre da zero.
4. Un indice negativo accede agli elementi a partire dalla fine della lista, contando all'indietro. L'ultimo elemento di una lista non vuota è sempre `a_list[-1]`.
5. Se l'indice negativo vi confonde, pensatela in questo modo: `a_list[-n] == a_list[len(a_list) - n]`. Quindi, in questa lista, `a_list[-3] == a_list[5 - 3] == a_list[2]`.

2.4.2. AFFETTARE UNA LISTA

Una volta che avete definito una lista, potete ottenerne una parte qualsiasi sotto forma di una nuova lista. Questa operazione si chiama *affettare* la lista.

```
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'esempio']
>>> a_list[1:3]           ①
['b', 'mpilgrim']
>>> a_list[1:-1]          ②
['b', 'mpilgrim', 'z']
>>> a_list[0:3]           ③
['a', 'b', 'mpilgrim']
>>> a_list[:3]            ④
['a', 'b', 'mpilgrim']
>>> a_list[3:]            ⑤
['z', 'esempio']
>>> a_list[:]             ⑥
['a', 'b', 'mpilgrim', 'z', 'esempio']
```

*a_list[0] è il
primo
elemento di
a_list.*

1. Potete ottenere una parte di una lista, chiamata “fetta”, specificando due indici. Il valore di ritorno è una nuova lista che contiene tutti gli elementi della lista, in ordine, a cominciare dal primo indice della fetta (in questo caso `a_list[1]`) fino al secondo indice della fetta escluso (in questo caso `a_list[3]`).
2. L'affettatura di una lista funziona anche se uno o entrambi gli indici della fetta sono negativi. Se vi è d'aiuto, potete pensarla in questo modo: leggendo la lista da sinistra a destra, il primo indice della fetta indica il primo elemento che volete, e il secondo indice della fetta indica il primo elemento che non volete. Il valore di ritorno è tutto quello che sta in mezzo.
3. L'indice delle liste comincia da zero, quindi `a_list[0:3]` restituisce i primi tre elementi della lista, cominciando da `a_list[0]` fino a `a_list[3]` escluso.
4. Se l'indice sinistro della fetta è 0, potete ometterlo e 0 diventa implicito. Quindi `a_list[:3]` è la stessa cosa di `a_list[0:3]`, perché lo 0 iniziale è implicito.
5. Similmente, se l'indice destro della fetta è la lunghezza della lista, potete ometterlo. Quindi `a_list[3:]` è la stessa cosa di `a_list[3:5]`, perché questa lista ha cinque elementi. C'è una piacevole simmetria qui. In questa lista di cinque elementi, `a_list[:3]` restituisce i primi 3 elementi e `a_list[3:]` restituisce gli ultimi due elementi. In effetti, `a_list[:n]` restituirà sempre i primi n elementi e `a_list[n:]` restituirà il resto, a prescindere dalla lunghezza della lista.
6. Se entrambi gli indici della fetta vengono omessi, tutti gli elementi della lista vengono inclusi. Ma questa non è la stessa lista della variabile `a_list` originale. È una nuova lista a cui capita di avere tutti gli stessi elementi. `a_list[:]` è una notazione abbreviata per creare la copia completa di una lista.

2.4.3. AGGIUNGERE ELEMENTI A UNA LISTA


Ci sono quattro modi per aggiungere un elemento a una lista.

```

>>> a_list = ['a']
>>> a_list = a_list + [2.0, 3]           ①
>>> a_list                               ②
['a', 2.0, 3]
>>> a_list.append(True)                 ③
>>> a_list
['a', 2.0, 3, True]
>>> a_list.extend(['quattro', 'Ω'])    ④
>>> a_list
['a', 2.0, 3, True, 'quattro', 'Ω']
>>> a_list.insert(0, 'Ω')               ⑤
>>> a_list
['Ω', 'a', 2.0, 3, True, 'quattro', 'Ω']

```

1. L'operatore + concatena liste per creare una nuova lista. Una lista può contenere qualsiasi numero di elementi, in quanto non c'è alcun limite sulle dimensioni (a parte la memoria disponibile). Tuttavia, se la memoria è un problema, sappiate che la concatenazione tra liste crea una seconda lista in memoria. In questo caso, quella nuova lista viene immediatamente assegnata alla variabile `a_list` esistente. Quindi, questa riga di codice in realtà è un'operazione composta da due passi — concatenazione e poi assegnamento — che potrebbe (temporaneamente) consumare molta memoria quando avete a che fare con liste di grandi dimensioni.
2. Una lista può contenere elementi di ogni tipo di dato e gli elementi contenuti in una singola lista non devono per forza essere tutti dello stesso tipo. Qui abbiamo una lista contenente una stringa, un numero in virgola mobile e un intero.
3. Il metodo `append()` aggiunge un singolo elemento alla fine di una lista. (Ora abbiamo *quattro* tipi di dato differenti nella lista!)
4. Le liste sono implementate come classi. “Creare” una lista vuol dire in realtà istanziare una classe. Come tale, una lista è dotata di metodi che operano su di essa. Il metodo `extend()` prende una lista come unico argomento e aggiunge ognuno degli elementi dell'argomento in coda alla lista originale.
5. Il metodo `insert()` inserisce un singolo elemento in una lista. Il primo argomento è l'indice del primo elemento nella lista che verrà spostato dalla sua posizione. Gli elementi di una lista non devono essere unici; per esempio, ora ci sono due elementi diversi con il valore 'a', `a_list[0]` e `a_list[1]`.

 `a_list.insert(0, valore)` ha lo stesso effetto della funzione `unshift()` in Perl.

Aggiunge un elemento all'inizio della lista e incrementa l'indice posizionale di tutti gli altri elementi per fare spazio.

Diamo un'occhiata più da vicino alle differenze tra `append()` ed `extend()`.

```
>>> a_list = ['a', 'b', 'c']
>>> a_list.extend(['d', 'e', 'f']) ①
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(a_list) ②
6
>>> a_list[-1]
'f'
>>> a_list.append(['g', 'h', 'i']) ③
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]
>>> len(a_list) ④
7
>>> a_list[-1]
['g', 'h', 'i']
```

1. Il metodo `extend()` prende un singolo argomento, che è sempre una lista, e aggiunge ogni elemento di quella lista ad `a_list`.
2. Se cominciate con una lista di tre elementi e la estendete con una lista di altri tre elementi, terminerete con una lista di sei elementi.
3. D'altra parte, il metodo `append()` prende un singolo argomento, che può avere un tipo di dato qualsiasi. Qui, state chiamando il metodo `append()` con una lista di tre elementi.
4. Se cominciate con una lista di sei elementi e vi accodate una lista, terminerete con... una lista di sette elementi. Perché sette? Perché l'ultimo elemento (che avete appena aggiunto) è *anch'esso una lista*. Le liste possono contenere ogni tipo di dato, comprese altre liste. Questo potrebbe essere quello che desiderate, oppure no. Ma è quello che avete chiesto, ed è quello che avete ottenuto.

2.4.4. CERCARE VALORI IN UNA LISTA

```
>>> a_list = ['a', 'b', 'nuovo', 'mpilgrim', 'nuovo']
>>> a_list.count('nuovo')    ①
2
>>> 'nuovo' in a_list        ②
True
>>> 'c' in a_list
False
>>> a_list.index('mpilgrim')  ③
3
>>> a_list.index('nuovo')    ④
2
>>> a_list.index('c')        ⑤
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
```

1. Come potreste aspettarvi, il metodo `count()` restituisce il numero di occorrenze di uno specifico valore in una lista.
2. Nel caso tutto quello che vogliate sapere sia se un valore è presente nella lista oppure no, l'operatore `in` è leggermente più veloce rispetto al metodo `count()`. L'operatore `in` restituisce sempre `True` o `False`, senza dirvi quante volte il valore compare nella lista.
3. Né l'operatore `in` né il metodo `count()` vi diranno *dove* si trova il valore. Se avete bisogno di questa informazione, chiamate il metodo `index()`. Per default questo metodo cercherà nell'intera lista, anche se potete specificare come secondo argomento opzionale l'indice (a partire da 0) da cui cominciare, e come terzo argomento opzionale persino l'indice (a partire da 0) in cui terminare la ricerca.
4. Il metodo `index()` trova la *prima* occorrenza di un valore nella lista. In questo caso, `'nuovo'` appare due volte nella lista, in `a_list[2]` e `a_list[4]`, ma il metodo `index()` restituirà solamente l'indice della prima occorrenza.
5. Come *non* vi sareste aspettati, se il valore non viene trovato nella lista il metodo `index()` solleverà un'eccezione.

Siete sorpresi? Ma è così: il metodo `index()` solleva un'eccezione se non trova il valore nella lista. Questa è una differenza considerevole rispetto alla maggior parte dei linguaggi, che restituiscono un qualche indice non valido (come -1). Sebbene questo possa sembrare fastidioso all'inizio, penso che finirete per apprezzarlo. Significa che il vostro programma fallirà alla sorgente del problema invece di fallire stranamente e silenziosamente più tardi. Ricordate, -1 è un indice di lista valido. Se il metodo `index()` restituisse -1, questo potrebbe condurre ad alcune sessioni di debug non esattamente divertenti!

2.4.5. RIMUOVERE ELEMENTI DA UNA LISTA

Le liste possono espandersi e contrarsi automaticamente. Finora avete visto la parte relativa all'espansione. Esistono anche diversi modi per rimuovere elementi da una lista.

*Le liste non
hanno mai
buchi.*

```
>>> a_list = ['a', 'b', 'nuovo', 'mpilgrim', 'nuovo']
>>> a_list[1]
'b'
>>> del a_list[1]           ①
>>> a_list
['a', 'nuovo', 'mpilgrim', 'nuovo']
>>> a_list[1]             ②
'nuovo'
```

1. Potete usare l'istruzione `del` per cancellare uno specifico elemento da una lista.

2. Accedere all'elemento di indice 1 dopo aver cancellato l'elemento di indice 1 *non* risulta in un errore. L'indice posizionale di tutti gli elementi che seguono l'elemento rimosso verrà spostato per “riempire il buco” lasciato dalla cancellazione dell'elemento.

Non conoscete l'indice posizionale? Non è un problema, perché potete anche rimuovere gli elementi tramite il loro valore.

```
>>> a_list.remove('nuovo') ①
>>> a_list
['a', 'mpilgrim', 'nuovo']
>>> a_list.remove('nuovo') ②
>>> a_list
['a', 'mpilgrim']
>>> a_list.remove('nuovo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

1. Potete anche rimuovere un elemento da una lista con il metodo `remove()`. Il metodo `remove()` prende un *valore* e rimuove la prima occorrenza di quel valore dalla lista. Ancora una volta, l'indice posizionale di tutti gli elementi che seguono l'elemento cancellato verrà diminuito per “riempire il buco”. Le liste non hanno mai buchi.
2. Potete chiamare il metodo `remove()` tutte le volte che volete, ma verrà sollevata un'eccezione se provate a rimuovere un valore che non è presente nella lista.

2.4.6. RIMUOVERE ELEMENTI DA UNA LISTA: GIRO BONUS

Un altro metodo interessante delle liste è `pop()`. Il metodo `pop()` rappresenta ancora un altro modo di rimuovere elementi da una lista, ma con una peculiarità.

```

>>> a_list = ['a', 'b', 'nuovo', 'mpilgrim']
>>> a_list.pop() ①
'mpilgrim'
>>> a_list
['a', 'b', 'nuovo']
>>> a_list.pop(1) ②
'b'
>>> a_list
['a', 'nuovo']
>>> a_list.pop()
'nuovo'
>>> a_list.pop()
'a'
>>> a_list.pop() ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list

```

1. Quando viene invocato senza argomenti, il metodo `pop()` delle liste rimuove l'ultimo elemento nella lista e restituisce il valore che ha rimosso.
2. Potete rimuovere elementi arbitrari da una lista passando un indice posizionale al metodo `pop()`. Il metodo rimuoverà quell'elemento, sposterà tutti gli elementi seguenti per “riempire il buco” e restituirà il valore che ha rimosso.
3. Invocare `pop()` su una lista vuota solleva un'eccezione.



Chiamare il metodo `pop()` delle liste senza un argomento ha lo stesso effetto della funzione `pop()` in Perl: rimuove l'ultimo elemento da una lista e restituisce il valore dell'elemento rimosso. Perl possiede un'altra funzione, `shift()`, che rimuove il primo elemento e restituisce il suo valore; in Python, questa operazione è equivalente a `a_list.pop(0)`.

2.4.7. LISTE IN UN CONTESTO LOGICO

Potete usare una lista anche in un contesto logico come quello di un'istruzione `if`.

```
>>> def is_it_true(anything):
...     if anything:
...         print('sì, è vero')
...     else:
...         print('no, è falso')
...
>>> is_it_true([])           ①
no, è falso
>>> is_it_true(['a'])       ②
sì, è vero
>>> is_it_true([False])     ③
sì, è vero
```

*Le liste vuote
sono false;
tutte le altre
liste sono
vere.*

1. In un contesto logico, una lista vuota è falsa.
2. Qualsiasi lista con almeno un elemento è vera.
3. Qualsiasi lista con almeno un elemento è vera. Il valore degli elementi è irrilevante.

*
**

2.5. TUPLE

Una tupla è una lista immutabile. Una tupla non può essere modificata in alcun modo dopo che è stata creata.

```

>>> a_tuple = ("a", "b", "mpilgrim", "z", "esempio") ①
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'esempio')
>>> a_tuple[0] ②
'a'
>>> a_tuple[-1] ③
'esempio'
>>> a_tuple[1:3] ④
('b', 'mpilgrim')

```

1. Una tupla si definisce nello stesso modo di una lista, a parte il fatto che l'intero insieme di elementi è racchiuso tra parentesi tonde anziché parentesi quadre.
2. Gli elementi di una tupla hanno un ordine definito, esattamente come in una lista. Gli indici di una tupla partono da zero, esattamente come in una lista, quindi il primo elemento di una tupla non vuota è sempre `a_tuple[0]`.
3. Gli indici negativi partono dalla fine della tupla, esattamente come in una lista.
4. Anche l'affettatura funziona esattamente come per una lista. Quando affettate una lista, ottenete una nuova lista; quando affettate una tupla, ottenete una nuova tupla.

La differenza principale tra tuple e liste è che le tuple non possono essere modificate. In termini tecnici, le tuple sono immutabili. In termini pratici, non sono dotate di alcun metodo che vi permetterebbe di modificarle. Le liste hanno metodi come `append()`, `extend()`, `insert()`, `remove()` e `pop()`. Le tuple sono prive di tutti questi metodi. Potete affettare una tupla (perché questa operazione crea una nuova tupla) e potete controllare se una tupla contiene un particolare valore (perché questa operazione non modifica la tupla) e... questo è tutto.

```

# continua dall'esempio precedente
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'esempio')
>>> a_tuple.append("nuovo") ①
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> a_tuple.remove("z") ②
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> a_tuple.index("esempio") ③
4
>>> "z" in a_tuple ④
True

```

1. Non potete aggiungere elementi a una tupla. Le tuple non hanno alcun metodo `append()` o `extend()`.
2. Non potete rimuovere elementi da una tupla. Le tuple non hanno alcun metodo `remove()` o `pop()`.
3. Potete cercare elementi in una tupla, dato che questa operazione non modifica la tupla.
4. Potete anche usare l'operatore `in` per controllare se un elemento è presente nella tupla.

Quindi a cosa servono le tuple?

- Le tuple sono più veloci delle liste. Se state definendo un insieme di valori costante e tutto quello per cui lo userete è iterare attraverso i suoi elementi, usate una tupla invece di una lista.
- Il vostro codice sarà più sicuro se “protegete dalla scrittura” i dati che non hanno bisogno di essere modificati. Usare una tupla invece di una lista è come avere un'istruzione `assert` implicita che mostra che questi dati sono costanti e che è necessario ricorrere a un ragionamento particolare (e a una funzione particolare) per eludere questo vincolo.
- Alcune tuple possono essere usate come chiavi di un dizionario (nello specifico, tuple che contengono valori *immutabili* come stringhe, numeri e altre tuple). Le liste non possono mai essere usate come chiavi di un dizionario, perché le liste non sono immutabili.



Le tuple possono essere convertite in liste e viceversa. La funzione predefinita `tuple()` accetta una lista e restituisce una tupla con gli stessi elementi; la funzione `list()` accetta una tupla e restituisce una lista. In effetti, `tuple()` congela una lista e `list()` scongela una tupla.

2.5.1. TUPLE IN UN CONTESTO LOGICO

Potete usare le tuple in un contesto logico come quello di un'istruzione `if`.

```
>>> def is_it_true(anything):
...     if anything:
...         print('sì, è vero')
...     else:
...         print('no, è falso')
...
>>> is_it_true(())           ①
no, è falso
>>> is_it_true(('a', 'b'))   ②
sì, è vero
>>> is_it_true((False,))     ③
sì, è vero
>>> type((False))           ④
<class 'bool'>
>>> type((False,))
<class 'tuple'>
```

1. In un contesto logico, una tupla vuota è falsa.
2. Qualsiasi tupla con almeno un elemento è vera.
3. Qualsiasi tupla con almeno un elemento è vera. Il valore degli elementi è irrilevante. Ma cosa ci fa lì quella virgola?
4. Per creare una tupla di un elemento, dovete inserire una virgola dopo il valore. Senza la virgola, Python presume che abbiate semplicemente inserito una coppia aggiuntiva di parentesi, che è innocua ma non crea una tupla.

2.5.2. ASSEGNARE PIÙ DI UN VALORE ALLA VOLTA

Ecco una fantastica scorciatoia di programmazione: in Python, potete usare una tupla per assegnare più di un valore alla volta.

```
>>> v = ('a', 2, True)
>>> (x, y, z) = v      ①
>>> x
'a'
>>> y
2
>>> z
True
```

1. v è una tupla di tre elementi e (x, y, z) è una tupla di tre variabili. Assegnare l'una all'altra assegna ognuno dei valori di v a ognuna delle variabili, in ordine.

Questa scorciatoia può essere usata in tutti i modi. Supponete di voler assegnare nomi a un intervallo di valori. Potete usare la funzione predefinita `range()` con l'assegnamento multivariabile per assegnare velocemente valori consecutivi.

```
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7) ①
>>> MONDAY                                ②
0
>>> TUESDAY
1
>>> SUNDAY
6
```

1. La funzione predefinita `range()` costruisce una sequenza di interi. (Tecnicamente, la funzione `range()` restituisce un iteratore, non una lista di tuple, ma imparerete la distinzione più avanti.) `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` e `SUNDAY` sono le variabili che state definendo. (Questo esempio proviene dal modulo `calendar`, un piccolo modulo divertente che stampa calendari, come il programma UNIX `cal`. Il modulo `calendar` definisce costanti intere per i giorni della settimana.)

2. Ora ogni variabile ha il suo valore: MONDAY vale 0, TUESDAY vale 1, e così via.

Potete anche usare l'assegnamento multivariabile per costruire funzioni che restituiscono più di un valore, semplicemente restituendo una tupla di tutti i valori. Il chiamante può trattarla come una singola tupla, oppure può assegnare i valori a singole variabili. Molte librerie Python standard lo fanno, incluso il modulo `os` che vedrete nel prossimo capitolo.



2.6. INSIEMI

Un insieme è un “gruppo” non ordinato di valori unici. Un singolo insieme può contenere valori di qualsiasi tipo di dato immutabile. Una volta che avete due insiemi, potete effettuare le classiche operazioni sugli insiemi come l'unione, l'intersezione e la differenza tra insiemi.

2.6.1. CREARE UN INSIEME

Cominciamo dall'inizio. Creare un insieme è facile.

```
>>> a_set = {1}      ①
>>> a_set
{1}
>>> type(a_set)      ②
<class 'set'>
>>> a_set = {1, 2}   ③
>>> a_set
{1, 2}
```

1. Per creare un insieme con un valore, mettete il valore tra parentesi graffe (`{}`).
2. Gli insiemi sono effettivamente implementati come classi, ma non preoccupatevi di questo per ora.
3. Per creare un insieme con più valori, separate i valori con virgole e circondate il tutto con parentesi graffe.

Potete anche creare un insieme a partire da una lista.

```
>>> a_list = ['a', 'b', 'mpilgrim', True, False, 42]
>>> a_set = set(a_list) ①
>>> a_set ②
{'a', False, 'b', True, 'mpilgrim', 42}
>>> a_list ③
['a', 'b', 'mpilgrim', True, False, 42]
```

1. Per creare un insieme da una lista, usate la funzione `set()`. (I pedanti che sanno come sono implementati gli insiemi vi faranno notare che questa non è esattamente l'invocazione di una funzione, ma la creazione di un'istanza di una classe. Vi *prometto* che imparerete la differenza più avanti in questo libro, ma per ora vi basti sapere che `set()` agisce come una funzione e restituisce un insieme.)
2. Come ho menzionato in precedenza, un singolo insieme può contenere valori di qualsiasi tipo di dato. E, come ho menzionato in precedenza, gli insiemi *non sono ordinati*. Questo insieme non ricorda l'ordine originale della lista che è stata usata per crearlo. Se doveste aggiungere elementi a questo insieme, non ricorderebbe l'ordine in cui li avete aggiunti.
3. La lista originale rimane invariata.

Non avete ancora alcun valore? Non è un problema. Potete creare un insieme vuoto.

```
>>> a_set = set() ①
>>> a_set ②
set()
>>> type(a_set) ③
<class 'set'>
>>> len(a_set) ④
0
>>> not_sure = {} ⑤
>>> type(not_sure)
<class 'dict'>
```

1. Per creare un insieme vuoto, chiamate `set()` senza argomenti.

2. La rappresentazione stampata di un insieme vuoto ha un'aria un po' strana. Vi sareste aspettati {}, forse? Ma quella rappresentazione denoterebbe un dizionario vuoto. Imparerete cosa sono i dizionari più avanti in questo capitolo.
3. Nonostante la strana rappresentazione stampata, questo è un insieme...
4. ...e questo insieme non ha elementi.
5. A causa di alcuni cavilli storicamente legati a Python 2, non potete creare un insieme vuoto con due parentesi graffe. Questa notazione in realtà crea un dizionario vuoto, non un insieme vuoto.

2.6.2. MODIFICARE UN INSIEME

Esistono due modi differenti per aggiungere valori a un insieme esistente: il metodo `add()` e il metodo `update()`.

```
>>> a_set = {1, 2}
>>> a_set.add(4) ①
>>> a_set
{1, 2, 4}
>>> len(a_set) ②
3
>>> a_set.add(1) ③
>>> a_set
{1, 2, 4}
>>> len(a_set) ④
3
```

1. Il metodo `add()` prende un singolo argomento, che può essere di qualsiasi tipo di dato, e aggiunge il valore passato all'insieme.
2. Ora questo insieme possiede 3 membri.
3. Gli insiemi sono gruppi di *valori unici*. Se provate ad aggiungere un valore che è già contenuto nell'insieme, non succederà nulla. Non verrà sollevato un errore, è semplicemente un'operazione senza alcun effetto.
4. Questo insieme possiede *ancora* 3 membri.

```

>>> a_set = {1, 2, 3}
>>> a_set
{1, 2, 3}
>>> a_set.update({2, 4, 6}) ①
>>> a_set ②
{1, 2, 3, 4, 6}
>>> a_set.update({3, 6, 9}, {1, 2, 3, 5, 8, 13}) ③
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 13}
>>> a_set.update([10, 20, 30]) ④
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 20, 30}

```

1. Il metodo `update()` prende come argomento un insieme e aggiunge tutti i suoi membri all'insieme originale. È come se aveste chiamato il metodo `add()` con ognuno dei membri dell'insieme.
2. I valori duplicati vengono ignorati, dato che gli insiemi non possono contenere duplicati.
3. In realtà, potete chiamare il metodo `update()` con un numero qualsiasi di argomenti. Quando viene invocato con due insiemi, il metodo `update()` aggiunge tutti i membri di ogni insieme all'insieme originale (scartando i duplicati).
4. Il metodo `update()` può accettare oggetti di un certo numero di tipi di dato differenti, comprese le liste. Quando viene invocato con una lista, il metodo `update()` aggiunge tutti gli elementi della lista all'insieme originale.

2.6.3. RIMUOVERE ELEMENTI DA UN INSIEME

Esistono tre modi di rimuovere singoli elementi da un insieme. I primi due, `discard()` e `remove()`, hanno una sottile differenza.

```

>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set
{1, 3, 36, 6, 10, 45, 15, 21, 28}
>>> a_set.discard(10) ①
>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.discard(10) ②
>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.remove(21) ③
>>> a_set
{1, 3, 36, 6, 45, 15, 28}
>>> a_set.remove(21) ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 21

```

1. Il metodo `discard()` prende un singolo valore come argomento e rimuove quel valore dall'insieme.
2. Se invocate il metodo `discard()` con un valore che non è presente nell'insieme, non succederà nulla. Nessuna eccezione, è semplicemente un'operazione senza alcun effetto.
3. Anche il metodo `remove()` prende un singolo valore come argomento e rimuove quel valore dall'insieme.
4. Ecco la differenza: se il valore non è presente nell'insieme, il metodo `remove()` solleva un'eccezione di tipo `KeyError`.

Come le liste, gli insiemi sono dotati di un metodo `pop()`.

```

>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}

>>> a_set.pop() ①
1
>>> a_set.pop()
3
>>> a_set.pop()
36
>>> a_set
{6, 10, 45, 15, 21, 28}

>>> a_set.clear() ②
>>> a_set
set()

>>> a_set.pop() ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'

```

1. Il metodo `pop()` rimuove un singolo valore da un insieme e restituisce quel valore. Tuttavia, dato che gli insiemi non sono ordinati, non c'è nessun "ultimo" valore in un insieme, quindi non c'è alcun modo di controllare quale sia il valore che viene rimosso. Il processo è completamente arbitrario.
2. Il metodo `clear()` rimuove *tutti* i valori dall'insieme, lasciandovi con un insieme vuoto. Questo è equivalente a `a_set = set()`, che avrebbe creato un nuovo insieme vuoto e sovrascritto il precedente valore della variabile `a_set`.
3. Il tentativo di invocare `pop()` per rimuovere un valore da un insieme vuoto solleverà un'eccezione di tipo `KeyError`.

2.6.4. OPERAZIONI COMUNI SUGLI INSIEMI

Il tipo `set` di Python supporta diverse operazioni comuni sugli insiemi.

```

>>> a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}

>>> 30 in a_set ①
True

>>> 31 in a_set
False

>>> b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}

>>> a_set.union(b_set) ②
{1, 2, 195, 4, 5, 6, 8, 12, 76, 15, 17, 18, 3, 21, 30, 51, 9, 127}

>>> a_set.intersection(b_set) ③
{9, 2, 12, 5, 21}

>>> a_set.difference(b_set) ④
{195, 4, 76, 51, 30, 127}

>>> a_set.symmetric_difference(b_set) ⑤
{1, 3, 4, 6, 8, 76, 15, 17, 18, 195, 127, 30, 51}

```

1. Per verificare se un valore è un membro di un insieme, usate l'operatore `in`. Questo funziona come per le liste.
2. Il metodo `union()` restituisce un nuovo insieme contenente tutti gli elementi che sono *in uno o nell'altro* insieme.
3. Il metodo `intersection()` restituisce un nuovo insieme contenente tutti gli elementi che sono *in entrambi* gli insiemi.
4. Il metodo `difference()` restituisce un nuovo insieme contenente tutti gli elementi che sono in `a_set` ma non in `b_set`.
5. Il metodo `symmetric_difference()` restituisce un nuovo insieme contenente tutti gli elementi che sono *esattamente uno solo* degli insiemi.

Tre di questi metodi sono simmetrici.

```

# continua dall'esempio precedente

>>> b_set.symmetric_difference(a_set) ①
{3, 1, 195, 4, 6, 8, 76, 15, 17, 18, 51, 30, 127}

>>> b_set.symmetric_difference(a_set) == a_set.symmetric_difference(b_set) ②
True

>>> b_set.union(a_set) == a_set.union(b_set) ③
True

>>> b_set.intersection(a_set) == a_set.intersection(b_set) ④
True

>>> b_set.difference(a_set) == a_set.difference(b_set) ⑤
False

```

1. La differenza simmetrica tra `a_set` e `b_set` *sembra* diversa dalla differenza simmetrica tra `b_set` e `a_set`, ma ricordatevi che gli insiemi non sono ordinati. Due insiemi qualsiasi che contengono gli stessi valori (nessuno escluso) sono considerati uguali.
2. E questo è esattamente quello che accade qui. Non fatevi ingannare dalla rappresentazione stampata di questi insiemi mostrata dalla Shell Python. Contengono gli stessi valori, quindi sono uguali.
3. Anche l'unione tra due insiemi è simmetrica.
4. Anche l'intersezione tra due insiemi è simmetrica.
5. La differenza tra due insiemi non è simmetrica. Questo ha senso, perché l'operazione è analoga alla sottrazione di un numero da un altro. L'ordine degli operandi ha importanza.

Infine, ci sono alcune domande che potete porre agli insiemi.


```

>>> a_set = {1, 2, 3}
>>> b_set = {1, 2, 3, 4}
>>> a_set.issubset(b_set)    ①
True
>>> b_set.issuperset(a_set)  ②
True
>>> a_set.add(5)             ③
>>> a_set.issubset(b_set)
False
>>> b_set.issuperset(a_set)
False

```

1. `a_set` è un sottoinsieme di `b_set` — tutti i membri di `a_set` sono anche membri di `b_set`.
2. Ponendo la stessa domanda al contrario, `b_set` è un sovrainsieme di `a_set`, perché tutti i membri di `a_set` sono anche membri di `b_set`.
3. Appena aggiungete ad `a_set` un valore che non è presente in `b_set`, entrambi i metodi restituiscono `False`.

2.6.5. INSIEMI IN UN CONTESTO LOGICO

Potete usare gli insiemi in un contesto logico come quello di un'istruzione `if`.

```

>>> def is_it_true(anything):
...     if anything:
...         print('sì, è vero')
...     else:
...         print('no, è falso')
...
>>> is_it_true(set())    ①
no, è falso
>>> is_it_true({'a'})    ②
sì, è vero
>>> is_it_true({False})  ③
sì, è vero

```

1. In un contesto logico, un insieme vuoto è falso.
2. Qualsiasi insieme con almeno un elemento è vero.
3. Qualsiasi insieme con almeno un elemento è vero. Il valore degli elementi è irrilevante.



2.7. DIZIONARI

Un dizionario è un insieme non ordinato di coppie chiave-valore. Quando aggiungete una chiave a un dizionario, dovete anche aggiungere un valore per quella chiave. (Potete sempre cambiare il valore più tardi.) I dizionari Python sono ottimizzati per recuperare il valore quando conoscete la chiave, ma non viceversa.



Un dizionario in Python è come un hash in Perl 5. In Perl 5, le variabili che memorizzano un hash cominciano sempre con il carattere %. In Python, le variabili possono essere chiamate in qualsiasi modo, e Python tiene traccia del tipo di dato internamente.

2.7.1. CREARE UN DIZIONARIO

Creare un dizionario è facile. La sintassi è simile agli insiemi, ma invece di valori avete coppie chiave-valore. Una volta che avete creato un dizionario, potete recuperarne i valori attraverso le loro chiavi.

```

>>> a_dict = {'server': 'db.diveintopython3.org', 'database': 'mysql'} ①
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict['server'] ②
'db.diveintopython3.org'
>>> a_dict['database'] ③
'mysql'
>>> a_dict['db.diveintopython3.org'] ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'db.diveintopython3.org'

```

1. Per prima cosa, create un nuovo dizionario con due elementi e assegnatelo alla variabile `a_dict`. Ogni elemento è una coppia chiave-valore, e l'intero insieme di elementi è racchiuso tra parentesi graffe.
2. `'server'` è una chiave, e il suo valore associato, riferito da `a_dict['server']`, è `'db.diveintopython3.org'`.
3. `'database'` è una chiave, e il suo valore associato, riferito da `a_dict['database']`, è `'mysql'`.
4. Potete ottenere i valori attraverso le chiavi, ma non potete ottenere le chiavi attraverso i valori. Quindi, `a_dict['server']` è `'db.diveintopython3.org'`, ma `a_dict['db.diveintopython3.org']` solleva un'eccezione perché `'db.diveintopython3.org'` non è una chiave.

2.7.2. MODIFICARE UN DIZIONARIO

I dizionari non hanno alcun limite predefinito sulla propria dimensione. Potete aggiungere nuove coppie chiave-valore a un dizionario in ogni momento, oppure potete modificare il valore di una chiave esistente. Continuando dall'esempio precedente:

```

>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}

>>> a_dict['database'] = 'blog' ①

>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'blog'}

>>> a_dict['user'] = 'mark' ②

>>> a_dict ③
{'server': 'db.diveintopython3.org', 'user': 'mark', 'database': 'blog'}

>>> a_dict['user'] = 'dora' ④

>>> a_dict
{'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}

>>> a_dict['User'] = 'mark' ⑤

>>> a_dict
{'User': 'mark', 'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}

```

1. Non potete avere chiavi duplicate in un dizionario. Assegnare un valore a una chiave esistente cancellerà il vecchio valore.
2. Potete aggiungere nuove coppie chiave-valore in ogni momento. Questa sintassi è identica a quella per modificare valori già esistenti.
3. Il nuovo elemento del dizionario (chiave 'user', valore 'mark') appare nel mezzo. In effetti, è stata solamente una coincidenza che gli elementi apparissero in ordine nel primo esempio; è allo stesso modo una coincidenza che appaiano fuori ordine ora.
4. Assegnare un valore a una chiave esistente nel dizionario sostituisce semplicemente il vecchio valore con il nuovo.
5. Questo assegnamento reinsertirà 'mark' come valore della chiave 'user'? No! Guardate bene la chiave — c'è una U maiuscola in 'User'. Le chiavi di un dizionario sono sensibili alle maiuscole, quindi questa istruzione crea una nuova coppia chiave-valore, senza sovrascriverne una esistente. A voi potrebbe sembrare simile, ma per quanto riguarda Python la chiave è completamente diversa.

2.7.3. DIZIONARI A VALORI MISTI

I dizionari non funzionano solo con le stringhe. I valori in un dizionario possono appartenere a qualsiasi tipo di dato, compresi interi, booleani, oggetti arbitrari e persino altri dizionari. E all'interno di un singolo dizionario i valori non devono essere tutti dello stesso tipo, ma potete mescolarli a piacimento. Le chiavi di

un dizionario hanno maggiori restrizioni, ma possono essere stringhe, interi, e di pochi altri tipi. Potete mescolare anche i tipi di dato delle chiavi all'interno di un singolo dizionario.

In effetti, avete già visto un dizionario con chiavi e valori che non sono stringhe nel vostro primo programma Python.

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

Analizziamo questo oggetto nella shell interattiva.

```
>>> SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
...             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
>>> len(SUFFIXES)      ①
2
>>> 1000 in SUFFIXES    ②
True
>>> SUFFIXES[1000]      ③
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
>>> SUFFIXES[1024]      ④
['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']
>>> SUFFIXES[1000][3]   ⑤
'TB'
```

1. Similmente a quello che accade per le liste e gli insiemi, la funzione `len()` vi restituisce il numero di chiavi in un dizionario.
2. E come per le liste e gli insiemi, potete usare l'operatore `in` per verificare se una chiave specifica è definita in un dizionario.
3. `1000` è una chiave nel dizionario `SUFFIXES` e il suo valore è una lista di otto elementi (otto stringhe, per essere precisi).
4. Similmente, `1024` è una chiave nel dizionario `SUFFIXES` e il suo valore è anch'esso una lista di otto elementi.
5. Dato che `SUFFIXES[1000]` è una lista, potete accedere ai singoli elementi contenuti in quella lista attraverso il loro indice a partire da 0.

2.7.4. DIZIONARI IN UN CONTESTO LOGICO

Potete usare i dizionari anche in un contesto logico come quello di un'istruzione `if`.

```
>>> def is_it_true(anything):
...     if anything:
...         print('sì, è vero')
...     else:
...         print('no, è falso')
...
>>> is_it_true({})                ①
no, è falso
>>> is_it_true({'a': 1})          ②
sì, è vero
```

1. In un contesto logico, un dizionario vuoto è falso.
2. Qualsiasi dizionario con almeno una coppia chiave-valore è vero.

*I dizionari
vuoti sono
falsi; tutti gli
altri dizionari
sono veri.*



2.8. None

`None` è una costante speciale in Python. È un valore nullo. `None` non è la stessa cosa di `False`. `None` non è 0. `None` non è una stringa vuota. Confrontare `None` con qualcos'altro che non sia `None` restituirà sempre `False`.

`None` è l'unico valore nullo. Ha un proprio tipo di dato (`NoneType`). Potete assegnare `None` a qualsiasi variabile, ma non potete creare altri oggetti di tipo `NoneType`. Tutte le variabili il cui valore è `None` sono uguali tra loro.

```

>>> type(None)
<class 'NoneType'>
>>> None == False
False
>>> None == 0
False
>>> None == ''
False
>>> None == None
True
>>> x = None
>>> x == None
True
>>> y = None
>>> x == y
True

```

2.8.1. None IN UN CONTESTO LOGICO

In un contesto logico, None è falso e not None è vero.

```

>>> def is_it_true(anything):
...     if anything:
...         print('sì, è vero')
...     else:
...         print('no, è falso')
...
>>> is_it_true(None)
no, è falso
>>> is_it_true(not None)
sì, è vero

```



2.9. LETTURE DI APPROFONDIMENTO

- Operazioni logiche
- Tipi numerici
- Tipi di sequenza
- Tipi di insieme
- Tipi di correlazione
- Il modulo `fractions`
- Il modulo `math`
- PEP 237: Unificare gli interi e gli interi lunghi
- PEP 238: Modificare l'operatore di divisione

CAPITOLO 3. DESCRIZIONI

“ La nostra immaginazione è tesa al massimo; non, come nelle storie fantastiche, per immaginare cose che in realtà non esistono, ma proprio per comprendere ciò che davvero esiste. ”

— Richard Feynman

3.1. IMMERSIONE!

Esaminando i linguaggi di programmazione, si nota che ognuno possiede una particolare funzionalità che, pur essendo complessa per natura, è stata resa intenzionalmente semplice. Quando state imparando un nuovo linguaggio potreste non riuscire a riconoscerla, perché i linguaggi che conoscete non la rendono altrettanto semplice, occupati come sono a semplificare qualche altra funzione. Questo capitolo vi insegnerà le descrizioni di lista, le descrizioni di dizionario e le descrizioni di insieme: tre concetti correlati basati su un'unica tecnica molto potente. Ma prima vorrei fare una breve digressione per parlarvi di due moduli che vi aiuteranno a navigare nel vostro file system locale.



3.2. LAVORARE CON I FILE E LE DIRECTORY

Python 3 include un modulo chiamato `os`, che sta per “sistema operativo”. Il modulo `os` contiene una pletora di funzioni per ottenere informazioni su — e, in alcuni casi, manipolare — directory locali, file, processi e variabili d'ambiente. Python fa del suo meglio per offrire una API unificata attraverso tutti i sistemi operativi supportati in modo che i vostri programmi possano funzionare su qualsiasi computer utilizzando la più piccola quantità possibile di codice specifico per la piattaforma.

3.2.1. LA DIRECTORY DI LAVORO CORRENTE

Se avete appena cominciato a lavorare con Python, vi ritroverete a passare molto tempo nella Shell Python. In tutto il libro, vedrete esempi che si sviluppano nel modo seguente.

1. Uno dei moduli contenuti nella cartella esempi viene importato.
2. Si invoca una funzione appartenente a quel modulo.
3. Il risultato della funzione viene spiegato.

Se non conoscete la directory di lavoro corrente, il passo n°1 probabilmente fallirà con un errore di tipo `ImportError`. Questo avviene perché Python cercherà il modulo di esempio nel percorso di ricerca per le importazioni, ma non lo troverà perché la cartella `esempi` non è una delle directory contenute nel percorso di ricerca. Per risolvere questo problema, avete due alternative.

1. Aggiungere la cartella `esempi` al percorso di ricerca per le importazioni.
2. Cambiare la directory di lavoro corrente impostandola alla cartella `esempi`.

La directory di lavoro corrente è una proprietà invisibile che Python mantiene in memoria tutto il tempo. C'è sempre una directory di lavoro corrente, sia che vi troviate nella Shell Python, o che eseguiate il vostro programma Python dalla riga di comando, oppure che invochiate uno script CGI Python su un server web in rete da qualche parte.

Il modulo `os` contiene due funzioni per lavorare con la directory di lavoro corrente.

*C'è sempre
una directory
di lavoro
corrente.*

```

>>> import os ①
>>> print(os.getcwd()) ②
C:\Python31
>>> os.chdir('/Users/pilgrim/diveintopython3/esempi') ③
>>> print(os.getcwd()) ④
C:\Users\pilgrim\diveintopython3\esempi

```

1. Il modulo `os` è incluso in Python, quindi potete importarlo ovunque e in ogni momento.
2. Usate la funzione `os.getcwd()` per ottenere la directory di lavoro corrente. Quando eseguite la Shell Python grafica, la directory di lavoro corrente viene impostata alla directory in cui si trova il file eseguibile della Shell Python. Su Windows, questa ubicazione dipende da dove avete installato Python; la directory di installazione predefinita è `C:\Python31`. Se eseguite la Shell Python dalla riga di comando, la directory di lavoro corrente viene impostata alla directory in cui vi trovavate quando avete invocato `python3`.
3. Usate la funzione `os.chdir()` per cambiare la directory di lavoro corrente.
4. Quando ho invocato la funzione `os.chdir()`, ho usato un percorso in stile Linux (con i caratteri di slash, senza la lettera del disco) anche se sono su Windows. Questo è uno dei casi in cui Python cerca di nascondere le differenze tra i sistemi operativi.

3.2.2. LAVORARE CON I NOMI DI FILE E DIRECTORY

Visto che stiamo parlando di directory voglio mostrarvi il modulo `os.path`, che contiene funzioni per manipolare nomi di file e directory.

```

>>> import os
>>> print(os.path.join('/Users/pilgrim/diveintopython3/esempi/', 'humansize.py')) ①
/Users/pilgrim/diveintopython3/esempi/humansize.py
>>> print(os.path.join('/Users/pilgrim/diveintopython3/esempi', 'humansize.py')) ②
/Users/pilgrim/diveintopython3/esempi\humansize.py
>>> print(os.path.expanduser('~')) ③
c:\Users\pilgrim
>>> print(os.path.join(os.path.expanduser('~'), 'diveintopython3', 'esempi', 'humansize.py')) ④
c:\Users\pilgrim\diveintopython3\esempi\humansize.py

```

1. La funzione `os.path.join()` costruisce un nome di percorso a partire da uno o più nomi di percorso parziali. In questo caso, non fa altro che concatenare stringhe.
2. In questo caso leggermente meno elementare, la funzione `os.path.join()` aggiungerà un carattere di slash ulteriore al nome di percorso prima di unirlo al nome del file. Il carattere è un backslash anziché uno slash, perché ho realizzato questo esempio su Windows. Se replicate questo esempio su Linux o Mac OS X, vedrete uno slash. Evitate di armeggiare con i caratteri di slash: usate sempre `os.path.join()` e lasciate che sia Python a scegliere quelli giusti.
3. La funzione `os.path.expanduser()` espanderà un nome di percorso che usa il carattere `~` per rappresentare la directory personale dell'utente corrente. Questa operazione funziona su tutte le piattaforme dove gli utenti hanno una directory personale, comprese Linux, Mac OS X e Windows. Il percorso restituito non ha uno slash finale, ma la funzione `os.path.join()` non se ne preoccupa.
4. Combinando queste tecniche, potete facilmente costruire nomi di percorso per directory e file che si trovano nella directory personale dell'utente. La funzione `os.path.join()` può accettare un numero qualsiasi di argomenti. Ero felicissimo quando l'ho scoperto, perché `aggiungiSlashSeNecessario()` è una di quelle piccole funzioni stupide che mi tocca sempre scrivere quando costruisco il mio armamentario di utilità in un nuovo linguaggio. *Non* dovete scrivere questa piccola funzione stupida in Python, perché alcune persone intelligenti se ne sono già occupate per voi.

Il modulo `os.path` contiene anche funzioni per suddividere nomi di percorso completi, nomi di directory e nomi di file nelle loro parti costituenti.

```
>>> pathname = '/Users/pilgrim/diveintopython3/esempi/humansize.py'
>>> os.path.split(pathname)                                ①
('/Users/pilgrim/diveintopython3/esempi', 'humansize.py')
>>> (dirname, filename) = os.path.split(pathname)          ②
>>> dirname                                                ③
'/Users/pilgrim/diveintopython3/esempi'
>>> filename                                              ④
'humansize.py'
>>> (shortname, extension) = os.path.splitext(filename)   ⑤
>>> shortname
'humansize'
>>> extension
'.py'
```

1. La funzione `split()` suddivide un nome di percorso completo e restituisce una tupla contenente il percorso e il nome di file.
2. Ricordate quando ho detto che potevate usare un assegnamento multivariabile per restituire più di un valore da una funzione? La funzione `os.path.split()` fa esattamente questo. Se assegnate il valore restituito dalla funzione `split()` a una tupla di due variabili, ogni variabile riceve il valore del corrispondente elemento contenuto nella tupla restituita.
3. La prima variabile, `dirname`, riceve il primo elemento della tupla restituita dalla funzione `os.path.split()`, cioè il percorso del file.
4. La seconda variabile, `filename`, riceve il valore del secondo elemento della tupla restituita dalla funzione `os.path.split()`, cioè il nome del file.
5. Il modulo `os.path` contiene anche la funzione `os.path.splitext()`, che suddivide un nome di file e restituisce una tupla contenente il nome del file e l'estensione del file. Potete usare la tecnica già vista per assegnare ogni valore a variabili separate.

3.2.3. ELENCARE IL CONTENUTO DELLE DIRECTORY

Il modulo `glob` è un altro strumento incluso nella libreria standard di Python. Vi fornisce un modo facile per ottenere programmaticamente il contenuto di una directory e usa quelle combinazioni di caratteri chiamate *wildcard* che potreste aver già visto lavorando sulla riga di comando.

*Il modulo
glob usa le
wildcard*

```
>>> os.chdir('/Users/pilgrim/diveintopython3/')
>>> import glob
```

*tipiche della
shell.*

```
>>> glob.glob('esempi/*.xml') ①
['esempi\\feed-broken.xml',
 'esempi\\feed-ns0.xml',
 'esempi\\feed.xml']

>>> os.chdir('esempi/') ②
>>> glob.glob('*test*.py') ③
['alphameticstest.py',
 'pluraltest1.py',
 'pluraltest2.py',
 'pluraltest3.py',
 'pluraltest4.py',
 'pluraltest5.py',
 'pluraltest6.py',
 'romantest1.py',
 'romantest10.py',
 'romantest2.py',
 'romantest3.py',
 'romantest4.py',
 'romantest5.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```

1. Il modulo `glob` prende una wildcard e restituisce il percorso di tutti i file e le directory che corrispondono alla wildcard. In questo esempio, la wildcard è un percorso di directory più `*.xml`, che corrisponde a tutti i file `.xml` nella sottodirectory `esempi`.
2. Ora impostiamo la directory di lavoro corrente alla sottodirectory `esempi`. La funzione `os.chdir()` può accettare nomi di percorso relativi.
3. Potete includere molteplici wildcard nel vostro pattern di tipo `glob`. Questo esempio trova tutti i file nella directory di lavoro corrente che terminano con l'estensione `.py` e contengono la parola `test` da qualche parte nel loro nome.

3.2.4. OTTENERE METADATI SUI FILE

Tutti i file system moderni memorizzano alcuni metadati su ogni file: la data di creazione, la data dell'ultima modifica, la dimensione del file, e così via. Python vi fornisce una singola API per accedere a questi metadati. Non avete bisogno di aprire il file, tutto quello che vi serve è il suo nome.

```
>>> import os

>>> print(os.getcwd())           ①
c:\Users\pilgrim\diveintopython3\esempi

>>> metadata = os.stat('feed.xml') ②

>>> metadata.st_mtime             ③
1247520344.9537716

>>> import time                  ④

>>> time.localtime(metadata.st_mtime) ⑤
time.struct_time(tm_year=2009, tm_mon=7, tm_mday=13, tm_hour=17,
tm_min=25, tm_sec=44, tm_wday=0, tm_yday=194, tm_isdst=1)
```

1. La directory di lavoro corrente è la cartella `esempi`.
2. `feed.xml` è un file che si trova nella cartella `esempi`. Invocare la funzione `os.stat()` restituisce un oggetto che contiene diversi tipi di metadati sul file.
3. `st_mtime` è la data di modifica, ma è in un formato che non è molto utile. (Tecnicamente, rappresenta il numero di secondi trascorsi dal 1° gennaio 1970. Non sto scherzando.)
4. Il modulo `time` è parte della libreria standard di Python e contiene funzioni di conversione tra differenti rappresentazioni temporali, funzioni per formattare i valori temporali in stringhe e funzioni per lavorare con i fusi orari.

5. La funzione `time.localtime()` converte un valore temporale espresso come numero di secondi trascorsi dal 1° gennaio 1970 (come rappresentato dalla proprietà `st_mtime` restituita dalla funzione `os.stat()`) in una struttura più utile contenente anno, mese, giorno, ora, minuto, secondo, e così via. La modifica più recente a questo file è stata effettuata il 13 giugno 2009, attorno alle 17:25.

```
# continua dall'esempio precedente

>>> metadata.st_size ①
3070

>>> import humansize

>>> humansize.approximate_size(metadata.st_size) ②
'3.0 KiB'
```

1. La funzione `os.stat()` restituisce anche la dimensione di un file, nella proprietà `st_size`. Il file `feed.xml` è di 3070 byte.
2. Potete passare la proprietà `st_size` alla funzione `approximate_size()`.

3.2.5. COSTRUIRE NOMI DI PERCORSO ASSOLUTI

Nella sezione precedente avete visto come la funzione `glob.glob()` restituisca una lista di nomi di percorso relativi. Il primo esempio mostrava nomi di percorso come `'esempi\feed.xml'` e il secondo esempio mostrava nomi di percorso relativi ancora più brevi come `'romantest1.py'`. Fino a quando rimanete nella stessa directory di lavoro corrente, questi nomi di percorso relativi funzioneranno per aprire file e ottenere metadati sui file. Ma se volete costruire un nome di percorso assoluto — cioè uno che includa i nomi di tutte le directory fino alla directory radice o alla lettera del disco — allora avrete bisogno della funzione `os.path.realpath()`.

```
>>> import os

>>> print(os.getcwd())
c:\Users\pilgrim\diveintopython3\esempi

>>> print(os.path.realpath('feed.xml'))
c:\Users\pilgrim\diveintopython3\esempi\feed.xml
```



3.3. DESCRIZIONI DI LISTA

Una descrizione di lista fornisce un modo compatto di correlare una lista con un'altra lista applicando una funzione a ogni elemento della prima lista.

```
>>> a_list = [1, 9, 8, 4]
>>> [elem * 2 for elem in a_list] ①
[2, 18, 16, 8]
>>> a_list ②
[1, 9, 8, 4]
>>> a_list = [elem * 2 for elem in a_list] ③
>>> a_list
[2, 18, 16, 8]
```

1. Per capire il senso di questa espressione, esaminatela da destra verso sinistra. `a_list` è la lista di partenza. L'interprete Python percorre `a_list` un elemento alla volta, assegnando temporaneamente il valore di ogni elemento alla variabile `elem`. Poi, Python applica la funzione `elem * 2` e aggiunge quel risultato in coda alla lista restituita.
2. Una descrizione di lista crea una nuova lista e non modifica la lista originale.
3. Assegnare il risultato di una descrizione di lista alla variabile che contiene la lista di partenza non crea problemi. Python costruisce la nuova lista in memoria e quando la descrizione di lista è completa ne assegna il risultato alla variabile originale.

Una descrizione di lista può contenere un'espressione Python qualsiasi, incluse le funzioni del modulo `os` per manipolare file e directory.

*Una
descrizione di
lista può
contenere una
espressione
Python
qualsiasi.*

```
>>> import os, glob

>>> glob.glob('*.xml') ①
['feed-broken.xml', 'feed-ns0.xml', 'feed.xml']

>>> [os.path.realpath(f) for f in glob.glob('*.xml')] ②
['c:\\Users\\pilgrim\\diveintopython3\\esempi\\feed-broken.xml',
 'c:\\Users\\pilgrim\\diveintopython3\\esempi\\feed-ns0.xml',
 'c:\\Users\\pilgrim\\diveintopython3\\esempi\\feed.xml']
```

1. Questo restituisce una lista di tutti i file `.xml` nella directory di lavoro corrente.
2. Questa descrizione di lista prende la lista di file `.xml` e la trasforma in una lista di nomi di percorso completi.

Le descrizioni di lista possono anche filtrare gli elementi, producendo un risultato che può essere più piccolo della lista originale.

```
>>> import os, glob

>>> [f for f in glob.glob('*.py') if os.stat(f).st_size > 6000] ①
['pluraltest6.py',
 'romantest10.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```

1. Per filtrare una lista potete includere una clausola `if` in fondo alla descrizione di lista. L'espressione dopo la parola chiave `if` verrà valutata per ogni elemento della lista. Se l'espressione viene valutata a `True`, l'elemento verrà incluso nel risultato. Questa descrizione di lista esamina la lista di tutti i file `.py` nella directory corrente e l'espressione `if` filtra quella lista controllando se la dimensione di ogni file è più grande di 6000 byte. Ci sono sei file con questa caratteristica, quindi la descrizione di lista restituisce una lista di sei nomi di file.

Tutti gli esempi di descrizioni di lista visti finora hanno utilizzato espressioni semplici — moltiplica un numero per una costante, invoca una singola funzione, o semplicemente restituisci gli elementi della lista originale (dopo averli filtrati). Ma non c'è alcun limite alla complessità delle descrizioni di lista.

```

>>> import os, glob

>>> [(os.stat(f).st_size, os.path.realpath(f)) for f in glob.glob('*.xml')] ①
[(3074, 'c:\\Users\\pilgrim\\diveintopython3\\esempi\\feed-broken.xml'),
 (3386, 'c:\\Users\\pilgrim\\diveintopython3\\esempi\\feed-ns0.xml'),
 (3070, 'c:\\Users\\pilgrim\\diveintopython3\\esempi\\feed.xml')]

>>> import humansize

>>> [(humansize.approximate_size(os.stat(f).st_size), f) for f in glob.glob('*.xml')] ②
[('3.0 KiB', 'feed-broken.xml'),
 ('3.3 KiB', 'feed-ns0.xml'),
 ('3.0 KiB', 'feed.xml')]

```

1. Questa descrizione di lista trova tutti i file `.xml` nella directory di lavoro corrente, ottiene la dimensione di ogni file (invocando la funzione `os.stat()`) e costruisce una tupla contenente la dimensione del file e il percorso assoluto di ogni file (invocando la funzione `os.path.realpath()`).
2. Questa descrizione lavora a partire da quella precedente per invocare la funzione `approximate_size()` passando la dimensione di ogni file `.xml`.

*
**

3.4. DESCRIZIONI DI DIZIONARIO

Una descrizione di dizionario è come una descrizione di lista, ma costruisce un dizionario invece di una lista.

```

>>> import os, glob

>>> metadata = [(f, os.stat(f)) for f in glob.glob('*test*.py')] ①

>>> metadata[0] ②

('alphameticstest.py', nt.stat_result(st_mode=33206, st_ino=0, st_dev=0,
st_nlink=0, st_uid=0, st_gid=0, st_size=2509, st_atime=1247520344,
st_mtime=1247520344, st_ctime=1247520344))

>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*test*.py')} ③

>>> type(metadata_dict) ④

<class 'dict'>

>>> list(metadata_dict.keys()) ⑤

['romantest8.py', 'pluraltest1.py', 'pluraltest2.py', 'pluraltest5.py',
'pluraltest6.py', 'romantest7.py', 'romantest10.py', 'romantest4.py',
'romantest9.py', 'pluraltest3.py', 'romantest1.py', 'romantest2.py',
'romantest3.py', 'romantest5.py', 'romantest6.py', 'alphameticstest.py',
'pluraltest4.py']

>>> metadata_dict['alphameticstest.py'].st_size ⑥

2509

```

1. Questa non è una descrizione di dizionario, ma una descrizione di lista. Trova tutti i file .py il cui nome contiene la parola test, poi costruisce una tupla contenente il nome del file e i metadati del file (invocando la funzione `os.stat()`).
2. Ogni elemento della lista risultante è una tupla.
3. Questa è una descrizione di dizionario. La sintassi è simile a quella per una descrizione di lista, con due differenze. Primo, è racchiusa tra parentesi graffe anziché parentesi quadre. Secondo, invece di una singola espressione per ogni elemento, contiene due espressioni separate da un carattere di due punti. L'espressione prima dei due punti (f in questo esempio) è la chiave del dizionario, l'espressione dopo i due punti (`os.stat(f)` in questo esempio) è il valore.
4. Una descrizione di dizionario restituisce un dizionario.
5. Le chiavi di questo particolare dizionario sono semplicemente i nomi di file restituiti dall'invocazione di `glob.glob('*test*.py')`.
6. Il valore associato a ogni chiave è il valore restituito dalla funzione `os.stat()`. Questo significa che possiamo “cercare” un file tramite il nome per ottenere i suoi metadati. Una delle parti dei metadati è `st_size`, la dimensione del file. Il file `alphameticstest.py` è di 2509 byte.

Come con le descrizioni di lista, potete includere una clausola `if` in una descrizione di dizionario per filtrare la sequenza in ingresso sulla base di un'espressione che viene valutata per ogni elemento.

```
>>> import os, glob, humanize
>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*')} ①
>>> humanize_dict = {os.path.splitext(f)[0]:humanize.approximate_size(meta.st_size) \
...                  for f, meta in metadata_dict.items() if meta.st_size > 6000} ②
>>> list(humanize_dict.keys()) ③
['romantest9', 'romantest8', 'romantest7', 'romantest6', 'romantest10', 'pluraltest6']
>>> humanize_dict['romantest9'] ④
'6.5 KiB'
```

1. Questa descrizione di dizionario costruisce una lista di tutti i file nella directory di lavoro corrente (`glob.glob('*')`), ottiene i metadati di ogni file (`os.stat(f)`) e costruisce un dizionario le cui chiavi sono i nomi dei file e i cui valori sono i metadati di ogni file.
2. Questa descrizione di dizionario viene costruita a partire dalla descrizione precedente, filtrandola per escludere i file più piccoli di 6000 byte (`if meta.st_size > 6000`) e usando poi la lista filtrata per costruire un dizionario le cui chiavi sono i nomi dei file senza l'estensione (`os.path.splitext(f)[0]`) e i cui valori sono le dimensioni approssimate di ogni file (`humanize.approximate_size(meta.st_size)`).
3. Come avete visto nell'esempio precedente, ci sono sei file con le caratteristiche richieste, quindi ci sono sei file in questo dizionario.
4. Il valore corrispondente a ogni chiave è la stringa restituita dalla funzione `approximate_size()`.

3.4.1. ALTRA ROBA DIVERTENTE DA FARE CON LE DESCRIZIONI DI DIZIONARIO

Ecco un trucco da usare con le descrizioni di dizionario che un giorno potrebbe esservi utile: scambiare le chiavi e i valori di un dizionario.

```
>>> a_dict = {'a': 1, 'b': 2, 'c': 3}
>>> {value:key for key, value in a_dict.items()}
{1: 'a', 2: 'b', 3: 'c'}
```

Naturalmente, questo funziona solo se i valori del dizionario sono immutabili, come stringhe o tuple. Se provate a farlo con un dizionario che contiene liste, il trucco fallirà in maniera piuttosto vistosa.

```
>>> a_dict = {'a': [1, 2, 3], 'b': 4, 'c': 5}
>>> {value:key for key, value in a_dict.items()}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <dictcomp>
TypeError: unhashable type: 'list'
```

*
**

3.5. DESCRIZIONI DI INSIEME

Non va dimenticato che anche gli insiemi hanno la propria sintassi di descrizione. Questa sintassi è notevolmente simile a quella per le descrizioni di dizionario. L'unica differenza è che gli insiemi contengono solo valori invece di coppie chiave:valore.

```
>>> a_set = set(range(10))
>>> a_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> {x ** 2 for x in a_set} ①
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> {x for x in a_set if x % 2 == 0} ②
{0, 8, 2, 4, 6}
>>> {2**x for x in range(10)} ③
{32, 1, 2, 4, 8, 64, 128, 256, 16, 512}
```

1. Le descrizioni di insieme possono prendere in ingresso un insieme. Questa descrizione di insieme calcola i quadrati dell'insieme dei numeri da 0 a 9.
2. Come con le descrizioni di lista e di dizionario, le descrizioni di insieme possono contenere una clausola `if` per filtrare gli elementi prima di restituirli nell'insieme dei risultati.
3. Le descrizioni di insieme non hanno necessariamente bisogno di prendere in ingresso un insieme, ma possono accettare qualsiasi sequenza.



3.6. LETTURE DI APPROFONDIMENTO

- Il modulo `os`
- `os` — Accesso portabile alle funzioni specifiche dei sistemi operativi
- Il modulo `os.path`
- `os.path` — Manipolazione dei nomi di file indipendente dalla piattaforma
- Il modulo `glob`
- `glob` — Corrispondenze di pattern per i nomi di file
- Il modulo `time`
- `time` — Funzioni per manipolare l'orario
- Descrizioni di lista
- Descrizioni di lista annidate
- Tecniche di attraversamento

CAPITOLO 4. STRINGHE

“ Te lo dico in nome della nostra amicizia.

Il tuo alfabeto termina dove il mio inizia! ”

— Dr. Seuss, *On Beyond Zebra!*

4.1. ALCUNE COSE NOIOSE CHE AVETE BISOGNO DI SAPERE PRIMA DI POTERVI IMMERGERE

Forse pochi ne sono consapevoli, ma il testo è qualcosa di incredibilmente complicato. Considerate l'alfabeto, tanto per cominciare. La popolazione di Bougainville ha il più piccolo alfabeto del mondo; il loro alfabeto Rotokas è composto da sole 12 lettere: A, E, G, I, K, O, P, R, S, T, U e V. All'altra estremità dello spettro, lingue come il cinese, il giapponese e il coreano hanno migliaia di caratteri. L'inglese, ovviamente, ha 26 lettere — 52 se contate le maiuscole e le minuscole separatamente — più una manciata di `!@#$$%&` simboli di punteggiatura.

Chi parla di “testo” probabilmente pensa a “caratteri e simboli su uno schermo di computer”. Ma i computer non si occupano di caratteri e simboli; i computer maneggiano bit e byte. Ogni frammento di testo che avete mai visto sullo schermo di un computer si trova in realtà memorizzato in una particolare *codifica di carattere*. Parlando per sommi capi, la codifica di carattere fornisce una corrispondenza tra quello che vedete sul vostro schermo e quello che il vostro computer mantiene effettivamente in memoria e su disco. Ci sono molte codifiche di carattere diverse, alcune ottimizzate per particolari lingue come il russo o il cinese o l'inglese, e altre che possono essere usate per più di una lingua.

In realtà le cose sono molto più complicate di così. Diverse codifiche hanno in comune molti caratteri, ma ogni codifica può usare una diversa sequenza di byte per archiviare effettivamente quei caratteri in memoria o su disco. Quindi potete pensare alla codifica di carattere come a una sorta di chiave di decodifica. Ogni volta che qualcuno vi dà una sequenza di byte — un file, una pagina web, qualsiasi cosa — e afferma che si tratta di “testo”, avete bisogno di sapere quale codifica di carattere è stata usata in modo da poter decodificare i byte in caratteri. Se vi viene data la chiave sbagliata, o addirittura nessuna chiave, vi ritrovate

con il non invidiabile compito di scoprire il codice da soli. Probabilmente sceglierete quello sbagliato, e il risultato non sarà intelleggibile.

Sicuramente avrete visto pagine web con strani caratteri simili a punti interrogativi in posti dove dovrebbero trovarsi gli apostrofi. Di solito questo significa che l'autore della pagina non ha dichiarato correttamente la codifica dei caratteri, il vostro browser non ha potuto fare altro che tirare a indovinare, e il risultato è stato un misto di caratteri attesi e inattesi. In inglese è solo un fastidio; in altre lingue il risultato potrebbe essere completamente illeggibile.

Esistono codifiche di carattere per tutte le lingue più diffuse al mondo. Dato che ogni lingua è differente e che la memoria e lo spazio su disco sono stati storicamente costosi, ogni codifica di carattere è ottimizzata per una particolare lingua. Con questo voglio dire che ogni codifica usa gli stessi numeri (0–255) per rappresentare i caratteri di quella lingua. Per esempio, avrete probabilmente familiarità con la codifica ASCII, che memorizza i caratteri inglesi come numeri da 0 a 127. (65 è la “A” maiuscola, 97 è la “a” minuscola, &c.) L'inglese ha un alfabeto molto semplice, quindi può essere completamente espresso in meno di 128 numeri. Per quelli di voi che sanno contare in base 2, questo significa utilizzare 7 degli 8 bit contenuti in un byte.

Le lingue dell'Europa occidentale come il francese, lo spagnolo e il tedesco hanno più lettere dell'inglese. O, più precisamente, hanno lettere combinate con vari segni diacritici, come il carattere ñ in spagnolo. La codifica più comune per queste lingue è la CP-1252, anche chiamata “WINDOWS-1252” perché viene largamente utilizzata su Microsoft Windows. La codifica CP-1252 condivide caratteri con la codifica ASCII nell'intervallo 0–127, ma poi si espande nell'intervallo 128–255 per caratteri come la n-con-sopra-una-tilde (241), la u-con-sopra-due-punti (252), &c. Questa è ancora una codifica a singolo byte, comunque; il numero più elevato, 255, sta ancora in un byte.

Poi ci sono lingue come il cinese, il giapponese e il coreano che hanno talmente tanti caratteri da richiedere un insieme di caratteri a più byte. Cioè, ogni “carattere” è rappresentato da un numero a 2 byte tra 0 e

*Tutto quello
che pensavate
di sapere
sulle stringhe
è sbagliato.*

65535. Ma codifiche multibyte differenti hanno gli stessi problemi di codifiche a singolo byte differenti, e cioè che ognuna usa gli stessi numeri per indicare cose diverse. È solo che l'intervallo dei numeri è più ampio, perché ci sono molti più caratteri da rappresentare.

Questo andava quasi bene in un mondo senza reti, dove il “testo” era qualcosa che scrivevate voi stessi e di cui occasionalmente stampavate qualche copia. Non c’era molto “testo semplice”. I programmatori scrivevano il codice sorgente in ASCII e tutti gli altri usavano i word processor, ognuno dei quali definiva il proprio formato (non di testo) che teneva traccia delle informazioni sulla codifica di carattere insieme agli stili, &c. Le persone leggevano questi documenti con lo stesso word processor dell’autore originale, così ogni cosa funzionava, più o meno.

Considerate ora l’avvento di reti globali come l’email e il web. Un sacco di “testo semplice” svolazzante per il globo, scritto su un computer, trasmesso attraverso un secondo computer e ricevuto e visualizzato da un terzo computer. I computer possono vedere solo numeri, ma i numeri potrebbero rappresentare cose diverse. Oh no! Cosa fare? I sistemi dovettero essere progettati per trasportare le informazioni di codifica insieme a ogni frammento di “testo semplice”. Ricordatevi che è la chiave di decodifica a correlare i numeri leggibili dal computer con i caratteri leggibili dalle persone. Una chiave di decodifica mancante significa testo alterato, inintelligibile, o peggio.

Considerate ora il tentativo di memorizzare molti frammenti di testo nello stesso posto, come nella stessa tabella di un database che contiene tutte le email che avete mai ricevuto. Avete ancora bisogno di archiviare la codifica di carattere insieme a ogni frammento di testo in modo da poterlo visualizzare correttamente. Pensate sia difficile? Provate a fare una ricerca nel vostro database di posta elettronica, il che significa effettuare al volo conversioni tra molteplici codifiche. Non vi sembra divertente?

Ora considerate la possibilità di documenti scritti in più lingue, dove i caratteri presi da diverse lingue sono uno di fianco all’altro nello stesso documento. (Suggerimento: i programmi che provavano a farlo tipicamente usavano codici di escape per passare da una “modalità” all’altra. Puff, eccovi in modalità russa koi8-r, quindi 24I rappresenta Я; puff, ora siete in modalità Mac Greek, quindi 24I rappresenta ώ.) E naturalmente vorrete effettuare ricerche anche su *quei* documenti.

E ora piangete pure, perché tutto quello che pensavate di sapere sulle stringhe è sbagliato e il “testo semplice” non esiste.



4.2. UNICODE

Entra Unicode.

Unicode è un sistema progettato per rappresentare *qualsiasi* carattere proveniente da *qualsiasi* lingua. Unicode rappresenta ogni lettera, carattere, o ideogramma come un numero di 4 byte. Ogni numero rappresenta un unico carattere usato in almeno una delle lingue del mondo. (Non tutti i numeri sono utilizzati, ma ne vengono adoperati più di 65535, quindi 2 byte non sarebbero stati sufficienti.) I caratteri usati in più di una lingua corrispondono generalmente allo stesso numero, a meno che non ci sia una buona ragione etimologica per non farlo. In ogni caso, c'è esattamente 1 numero per carattere ed esattamente 1 carattere per numero. Ogni numero rappresenta sempre un solo carattere, quindi non ci sono “modalità” di cui tenere traccia. U+0041 è sempre 'A', anche se la vostra lingua non possiede alcuna 'A'.

A prima vista, sembra una grande idea. Una codifica per dominarle tutte. Più di una lingua per documento. Nessun “passaggio di modalità” per cambiare codifica a metà del testo. Ma l'ovvia domanda dovrebbe subito saltarvi in mente. Quattro byte? Per ogni singolo carattere? Sembra uno spreco orrendo, specialmente per lingue come l'inglese e lo spagnolo che necessitano di meno di un byte (256 numeri) per esprimere ogni possibile carattere. In effetti è uno spreco persino per lingue basate su ideogrammi (come il cinese) che non hanno mai bisogno di più di due byte per carattere.

Esiste una codifica Unicode che usa quattro byte per carattere. È chiamata UTF-32, perché 32 bit = 4 byte. UTF-32 è la codifica più semplice; prende ogni carattere Unicode (un numero di 4 byte) e rappresenta il carattere con quello stesso numero. Questo ha alcuni vantaggi, il più importante dei quali è la possibilità di accedere al carattere di posizione N in una stringa in tempo costante, perché il carattere di posizione N comincia al byte di posizione 4×N. Ha anche diversi svantaggi, il più ovvio dei quali è la necessità di usare quattro maledetti byte per memorizzare ogni maledetto carattere.

Anche se i caratteri Unicode sono molti, a quanto pare la maggior parte delle persone non ne userà nessuno al di là dei primi 65535. Quindi, esiste un'altra codifica Unicode, chiamata UTF-16 (perché 16 bit = 2 byte). UTF-16 codifica ogni carattere tra 0 e 65535 come due byte, e poi usa un qualche sporco trucco nel caso dobbiate effettivamente rappresentare i rari caratteri Unicode che si trovano nel “piano astrale” al di là di

65535. Il vantaggio più ovvio: UTF-16 è due volte più efficiente in termini di spazio rispetto a UTF-32, perché ogni carattere richiede solo due byte da memorizzare invece di quattro byte (a parte quelli per cui non è così). E potete ancora facilmente accedere al carattere di posizione N in una stringa in tempo costante, se ipotizzate che la stringa non includa alcun carattere del piano astrale, che è una buona ipotesi fino a quando non lo è più.

Ma sia UTF-32 che UTF-16 presentano anche altri svantaggi non banali. Sistemi computerizzati differenti memorizzano i singoli byte in modi differenti. Questo significa che il carattere U+4E2D potrebbe essere memorizzato in UTF-16 come 4E 2D oppure 2D 4E, a seconda che il sistema sia di tipo big-endian o little-endian. (Per UTF-32 sono possibili ancora più ordinamenti di byte.) Fino a quando i vostri documenti non lasciano mai il vostro computer, siete al sicuro — applicazioni differenti sullo stesso computer utilizzeranno sempre lo stesso ordine dei byte. Ma nel momento in cui volete trasferire documenti tra sistemi, forse su una qualche sorta di world wide web, avrete bisogno di un modo per indicare in quale ordine sono memorizzati i vostri byte. Altrimenti, il sistema ricevente non ha alcun modo di sapere se la sequenza di due byte 4E 2D rappresenta U+4E2D oppure U+2D4E.

Per risolvere questo problema, le codifiche Unicode multibyte definiscono un “Byte Order Mark”, uno speciale carattere non stampabile che potete includere all’inizio del vostro documento per indicare in quale ordine sono memorizzati i vostri byte. Per UTF-16, il Byte Order Mark è U+FEFF. Se ricevete un documento UTF-16 che comincia con i byte FF FE, sapete che l’ordine dei byte va in una certa direzione; se comincia con FE FF, sapete che l’ordine dei byte è quello inverso.

Tuttavia, UTF-16 non è esattamente l’ideale, specialmente se avete a che fare con un sacco di caratteri ASCII. Se ci pensate, persino una pagina web in cinese conterrà un sacco di caratteri ASCII — tutti gli elementi e gli attributi che circondano i caratteri stampabili cinesi. Essere in grado di accedere al carattere di posizione N in tempo costante è piacevole, ma rimane ancora l’irritante problema di quei caratteri del piano astrale, il che significa che non potete *garantire* che ogni carattere sia esattamente due byte, e quindi non potete *davvero* accedere al carattere di posizione N in tempo costante a meno che non manteniate un indice separato. E ragazzi, c’è sicuramente moltissimo testo ASCII nel mondo...

Altre persone hanno ponderato a lungo su questi problemi, e hanno trovato una soluzione:

UTF-8

UTF-8 è un sistema di codifica Unicode a *lunghezza variabile*. Cioè, caratteri differenti occupano un numero differente di byte. Per i caratteri ASCII (A-Z, &c.) UTF-8 usa solo un byte per carattere. In effetti, usa gli stessi identici byte; i primi 128 caratteri (0–127) in UTF-8 sono indistinguibili da quelli ASCII. I caratteri “Extended Latin” come ñ e ö finiscono per occupare due byte. (I byte non sono semplicemente *code point* Unicode come sarebbero in UTF-16, ma i bit vengono pesantemente manipolati in qualche modo.) Caratteri cinesi come 中 finiscono per occupare tre byte. I caratteri raramente usati del “piano astrale” occupano quattro byte.

Svantaggi: dato che ogni carattere può occupare un numero differente di byte, accedere al carattere di posizione N è una operazione di complessità $O(N)$ — cioè, più lunga è la stringa, più tempo ci vuole per trovare uno specifico carattere. In più, i bit devono essere manipolati per codificare i caratteri in byte e decodificare i byte in caratteri.

Vantaggi: la codifica dei caratteri ASCII comuni è super-efficiente. Per i caratteri Extended Latin non è peggiore rispetto alla codifica UTF-16 e per i caratteri cinesi è migliore rispetto a UTF-32. In più (e dovete fidarvi di me su questo, perché non vi mostrerò i calcoli), grazie alla natura esatta delle manipolazioni sui bit, non ci sono problemi di ordinamento dei byte. Un documento codificato in UTF-8 usa esattamente lo stesso flusso di byte su qualsiasi computer.



4.3. IMMERSIONE!

In Python 3, tutte le stringhe sono sequenze di caratteri Unicode. Non esistono stringhe Python codificate in UTF-8, o stringhe Python codificate in CP-1252. “Questa stringa è in UTF-8?” è una domanda non valida. UTF-8 è un modo di codificare caratteri come sequenze di byte. Se volete prendere una stringa e trasformarla in una sequenza di byte attraverso una particolare codifica di carattere, Python 3 vi può aiutare. Se volete prendere una sequenza di byte e trasformarla in una stringa, Python 3 può aiutarvi anche in questo caso. I byte non sono caratteri, i byte sono byte. I caratteri sono un’astrazione. Una stringa è una sequenza di quelle astrazioni.

```
>>> s = '深入 Python'    ①
>>> len(s)                ②
9
>>> s[0]                  ③
'深'
>>> s + ' 3'              ④
'深入 Python 3'
```

1. Per creare una stringa, racchiudetela tra apici. Le stringhe Python possono essere definite utilizzando apici (') oppure virgolette (").
2. La funzione built-in `len()` restituisce la lunghezza della stringa, cioè il numero di caratteri. Questa è la stessa funzione che utilizzate per trovare la lunghezza di una lista, di una tupla, di un insieme, o di un dizionario. Una stringa è come una lista di caratteri.
3. Esattamente come ottenete singoli elementi da una lista, potete ottenere singoli caratteri da una stringa usando la notazione a indice.
4. Esattamente come con le liste, potete concatenare le stringhe utilizzando l'operatore `+`.



4.4. FORMATTARE LE STRINGHE

Diamo un'altra occhiata a `humansize.py`:

*Le stringhe
possono
essere definite
utilizzando
apici oppure
virgolette.*

```

SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],           ①
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Converte la dimensione di un file in una forma leggibile.           ②

    Argomenti con nome:
    size -- dimensione del file in byte
    a_kilobyte_is_1024_bytes -- se True (default), usa multipli di 1024
                                se False, usa multipli di 1000

    Restituisce: stringa

    ...                                                                    ③

    if size < 0:
        raise ValueError('il numero non deve essere negativo')           ④

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)                       ⑤

    raise ValueError('numero troppo grande')

```

1. 'KB', 'MB', 'GB'... ognuna di quelle è una stringa.
2. Le docstring di una funzione sono stringhe. Questa docstring si estende su più righe, e quindi usa tre apici consecutivi all'inizio e alla fine della stringa.
3. Questi tre apici consecutivi concludono la docstring.
4. C'è un'altra stringa, passata all'eccezione come un messaggio di errore.
5. C'è una... ehi, che diavolo è quello?

Python 3 supporta la formattazione di valori in stringhe. Sebbene questo possa includere espressioni molto complicate, l'uso più basilare consiste nell'inserire un valore in una stringa con un singolo segnaposto.


```
>>> username = 'mark'

>>> password = 'PapayaWhip' ①

>>> "La password di {0} è {1}".format(username, password) ②

"La password di mark è PapayaWhip"
```

1. No, la mia password non è davvero PapayaWhip.
2. Ci sono un sacco di cose che stanno succedendo qui. Prima di tutto, c'è una chiamata di metodo su un letterale stringa. *Le stringhe sono oggetti* e gli oggetti hanno metodi. Secondo, l'intera espressione viene valutata come una stringa. Terzo, {0} e {1} sono *campi di sostituzione*, che vengono rimpiazzati dagli argomenti passati al metodo `format()`.

4.4.1. NOMI DI CAMPO COMPOSTI

L'esempio precedente mostra il caso più semplice, dove i campi di sostituzione sono semplicemente interi. I campi di sostituzione interi sono trattati come indici di posizione nella lista di argomenti del metodo `format()`. Questo significa che {0} è rimpiazzato dal primo argomento (`username` in questo caso), {1} è rimpiazzato dal secondo argomento (`password`), &c. Potete avere tanti indici di posizione quanti sono gli argomenti e potete avere tanti argomenti quanti ne volete. Ma i campi di sostituzione sono molto più potenti di così.

```
>>> import humanize

>>> si_suffixes = humanize.SUFFIXES[1000] ①

>>> si_suffixes

['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']

>>> '1000{0[0]} = 1{0[1]}'.format(si_suffixes) ②

'1000KB = 1MB'
```

1. Invece di chiamare una funzione qualsiasi nel modulo `humanize`, state recuperando una delle strutture dati che il modulo definisce: la lista dei suffissi (potenze-di-1000) del Sistema internazionale di unità di misura ("SI").
2. Questo sembra complicato, ma non lo è. {0} si riferirebbe al primo argomento passato al metodo `format()`, `si_suffixes`. Ma `si_suffixes` è una lista. Così {0[0]} si riferisce al primo elemento della lista che è il primo argomento passato al metodo `format()`: 'KB'. Similmente, {0[1]} si riferisce al secondo elemento della stessa

lista: 'MB'. Ogni cosa fuori dalle parentesi graffe — compreso 1000, il segno di uguale e gli spazi — non viene toccata. Il risultato finale è la stringa '1000KB = 1MB'.

Questo esempio mostra che *i campi di sostituzione possono accedere a elementi e proprietà di una struttura dati usando (quasi) la stessa sintassi di Python*. Quelli che abbiamo appena visto si chiamano *nomi di campo composti*. Per formare un nome di campo composto che sia valido, è possibile:

- passare una lista e accedere a un elemento della lista tramite indice (come nell'esempio precedente);
- passare un dizionario e accedere a un valore del dizionario tramite chiave;
- passare un modulo e accedere alle variabili e alle funzioni del modulo tramite nome;
- passare un'istanza di una classe e accedere alle proprietà e ai metodi dell'istanza tramite nome;
- utilizzare una qualsiasi combinazione dei nomi precedenti.

Giusto per impressionarvi, eccovi un esempio che combina tutti i nomi appena visti:

```
>>> import humanize
>>> import sys
>>> '1MB = 1000{0.modules[humanize].SUFFIXES[1000][0]}'.format(sys)
'1MB = 1000KB'
```

Funziona in questo modo.

- Il modulo `sys` mantiene informazioni sull'istanza dell'interprete Python attualmente in esecuzione. Visto che lo avete appena importato, potete passare il modulo `sys` stesso come argomento al metodo `format()`. Quindi il campo di sostituzione `{0}` si riferisce al modulo `sys`.

*{0} è
sostituito dal
1° argomento
di format().
{1} è
sostituito dal
2°.*

- `sys.modules` è un dizionario di tutti i moduli che sono stati importati in questa istanza dell'interprete Python. Le chiavi sono i nomi dei moduli sotto forma di stringhe, i valori sono gli oggetti modulo stessi. Quindi il campo di sostituzione `{0.modules}` si riferisce al dizionario dei moduli importati.
- `sys.modules['humansize']` è il modulo `humansize` che avete appena importato. Il campo di sostituzione `{0.modules[humansize]}` si riferisce al modulo `humansize`. Notate la leggera differenza nella sintassi. In vero codice Python, le chiavi del dizionario `sys.modules` sono stringhe; per riferirvi a esse, avete bisogno di mettere le virgolette attorno al nome del modulo (e.g. `'humansize'`). Ma nell'ambito di un campo di sostituzione le virgolette attorno al nome della chiave del dizionario vanno omesse (e.g. `humansize`). Per citare la [PEP 3101: Formattazione avanzata delle stringhe](#): “Le regole per riconoscere la chiave di un elemento sono molto semplici. Se comincia con una cifra, allora viene trattato come un numero, altrimenti viene usato come una stringa.”
- `sys.modules['humansize'].SUFFIXES` è il dizionario definito all'inizio del modulo `humansize`. Il campo di sostituzione `{0.modules[humansize].SUFFIXES}` si riferisce a quel dizionario.
- `sys.modules['humansize'].SUFFIXES[1000]` è una lista di suffissi SI: `['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']`. Quindi il campo di sostituzione `{0.modules[humansize].SUFFIXES[1000]}` si riferisce a quella lista.
- `sys.modules['humansize'].SUFFIXES[1000][0]` è il primo elemento nella lista dei suffissi SI: `'KB'`. Quindi, il campo di sostituzione completo `{0.modules[humansize].SUFFIXES[1000][0]}` è rimpiazzato dalla stringa di due caratteri `KB`.

4.4.2. SPECIFICHE DI FORMATO

Ma aspettate! C'è di più! Diamo un'altra occhiata a quella strana linea di codice in `humansize.py`:

```
if size < multiple:
    return '{0:.1f} {1}'.format(size, suffix)
```

`{1}` viene rimpiazzato dal secondo argomento passato al metodo `format()`, che è `suffix`. Ma che cos'è `{0:.1f}`? È due cose: `{0}`, che riconoscerete, e `:.1f`, che non riconoscerete. La seconda metà (dai due punti in poi compresi) definisce una *specifica di formato*, che raffina ulteriormente il modo in cui la variabile rimpiazzata dovrà essere formattata.



Le specifiche di formato vi permettono di manipolare il testo di sostituzione in una varietà di modi utili, come accade con la funzione `printf()` in C. Potete aggiungere blocchi di zeri o spazi, allineare stringhe, controllare la precisione dei decimali e persino convertire i numeri in esadecimali.

Nell'ambito di un campo di sostituzione, i due punti (:) segnano l'inizio della specifica di formato. La specifica di formato “.1” significa “arrotonda al decimale più vicino” (cioè mostra solo una cifra dopo il punto decimale). La specifica di formato “f” significa “numero in virgola fissa” (al contrario della notazione esponenziale o di qualche altra rappresentazione decimale). Quindi, data una dimensione `size` di 698.24 e un suffisso `suffix` di 'GB', la stringa formattata diventerà '698.3 GB', perché 698.24 viene arrotondato alla prima cifra decimale e il suffisso viene aggiunto dopo il numero.

```
>>> '{0:.1f} {1}'.format(698.24, 'GB')
'698.3 GB'
```

Per tutti i dettagli più intricati sulle specifiche di formato, consultate la sezione [Mini-linguaggio per le specifiche di formato](#) nella documentazione ufficiale di Python.



4.5. ALTRI METODI DI USO COMUNE PER LE STRINGHE

Oltre alla formattazione, le stringhe possono effettuare un certo numero di altre operazioni utili.

```

>>> s = '''Finished files are the re- ①
... sult of years of scientif-
... ic study combined with the
... experience of years.'''

>>> s.splitlines() ②
['Finished files are the re-',
 'sult of years of scientif-',
 'ic study combined with the',
 'experience of years.']

>>> print(s.lower()) ③
finished files are the re-
sult of years of scientif-
ic study combined with the
experience of years.

>>> s.lower().count('f') ④
6

```

1. Potete introdurre stringhe multiriga nella shell interattiva di Python. Una volta che avete cominciato una stringa su più righe con i tripli apici, vi basta premere INVIO e la shell interattiva vi permetterà di continuare la stringa. Digitare i tripli apici di chiusura termina la stringa, e il successivo INVIO eseguirà il comando (in questo caso, l'assegnamento della stringa a `s`).
2. Il metodo `splitlines()` prende una stringa multiriga e restituisce una lista di stringhe, una per ogni riga della stringa originale. Notate che i caratteri di ritorno a capo alla fine di ogni riga non sono inclusi.
3. Il metodo `lower()` converte l'intera stringa in minuscolo. (Similmente, il metodo `upper()` converte una stringa in maiuscolo.)
4. Il metodo `count()` conta il numero di occorrenze di una sottostringa. Sì, ci sono davvero sei “f” in quella frase!

Ecco un altro caso comune. Diciamo che avete una lista di coppie chiave-valore nella forma `chiave1=valore1&chiave2=valore2`, e volete separarle e creare un dizionario della forma `{chiave1: valore1, chiave2: valore2}`.

```

>>> query = 'user=pilgrim&database=master&password=PapayaWhip'
>>> a_list = query.split('&') ①
>>> a_list
['user=pilgrim', 'database=master', 'password=PapayaWhip']
>>> a_list_of_lists = [v.split('=', 1) for v in a_list if '=' in v] ②
>>> a_list_of_lists
[['user', 'pilgrim'], ['database', 'master'], ['password', 'PapayaWhip']]
>>> a_dict = dict(a_list_of_lists) ③
>>> a_dict
{'password': 'PapayaWhip', 'user': 'pilgrim', 'database': 'master'}

```

1. Il metodo `split()` delle stringhe prende un delimitatore come argomento obbligatorio e restituisce una lista di stringhe dividendo la stringa in corrispondenza del delimitatore. In questo caso, il delimitatore è un carattere di E commerciale, ma potrebbe essere qualsiasi cosa.
2. Ora abbiamo una lista di stringhe, ognuna contenente una chiave, seguita da un segno di uguale, seguito da un valore. Possiamo usare una descrizione di lista per iterare sull'intera lista e dividere ogni stringa in due stringhe in corrispondenza del primo segno di uguale. Il secondo argomento del metodo `split()` è opzionale e rappresenta il numero di suddivisioni che volete effettuare: 1 significa “effettua una sola suddivisione”, quindi il metodo `split()` restituirà una lista di due elementi. (In teoria, anche un valore potrebbe contenere un segno di uguale. Se avessimo usato `'chiave=valore=pippo'.split('=')`, avremmo ottenuto una lista di tre elementi, `['chiave', 'valore', 'pippo']`.)
3. Infine, Python può trasformare quella lista-di-liste in un dizionario semplicemente passandola alla funzione `dict()`.



L'esempio precedente somiglia molto al riconoscimento dei parametri di richiesta in un URL, ma in effetti il riconoscimento degli URL nella vita reale è molto più complicato di così. Se avete a che fare con i parametri di richiesta in un URL, vi conviene utilizzare la funzione `urllib.parse.parse_qs()`, che è in grado di gestire alcuni casi limite non banali.

4.5.1. AFFETTARE UNA STRINGA

Una volta che avete definito una stringa, potete ottenerne una parte qualsiasi sotto forma di una nuova stringa. Questa operazione si chiama *affettare* la stringa. Affettare le stringhe funziona esattamente allo stesso modo di affettare le liste, cosa che ha senso perché le stringhe sono solo sequenze di caratteri.

```
>>> a_string = 'Il tuo alfabeto termina dove il mio inizia!'
>>> a_string[7:15]           ①
'alfabeto'
>>> a_string[7:-7]          ②
'alfabeto termina dove il mio '
>>> a_string[0:2]           ③
'Il'
>>> a_string[:23]           ④
'Il tuo alfabeto termina'
>>> a_string[23:]           ⑤
' dove il mio inizia!'
```

1. Potete ottenere una parte di una stringa, chiamata “fetta”, specificando due indici. Il valore di ritorno è una nuova stringa che contiene tutti i caratteri della stringa, in ordine, a partire dal primo indice della fetta.
2. Come nell’affettare liste, potete usare indici negativi per affettare le stringhe.
3. L’indice delle stringhe comincia da zero, quindi `a_string[0:2]` restituisce i primi due caratteri della stringa, cominciando da `a_string[0]` fino a `a_string[2]` escluso.
4. Se l’indice sinistro della fetta è 0, potete ometterlo e 0 diventa implicito. Quindi `a_string[:23]` è la stessa cosa di `a_string[0:23]`, perché lo 0 iniziale è implicito.
5. Similmente, se l’indice destro della fetta è la lunghezza della stringa, potete ometterlo. Quindi `a_string[23:]` è la stessa cosa di `a_string[23:43]`, perché questa stringa ha 43 caratteri. C’è una piacevole simmetria qui. In questa stringa di 43 caratteri, `a_string[:23]` restituisce sempre i primi 23 caratteri e `a_string[23:]` restituisce tutto tranne i primi 23 caratteri. In effetti, `a_string[:n]` restituirà sempre i primi n caratteri e `a_string[n:]` restituirà il resto, a prescindere dalla lunghezza della stringa.



4.6. STRINGHE VS. BYTE

I byte sono byte; i caratteri sono un'astrazione. Una sequenza immutabile di caratteri Unicode si chiama *stringa*. Una sequenza immutabile di numeri-tra-0-e-255 si chiama oggetto *byte*.

```
>>> by = b'abcd\x65' ①
>>> by
b'abcde'
>>> type(by) ②
<class 'bytes'>
>>> len(by) ③
5
>>> by += b'\xff' ④
>>> by
b'abcde\xff'
>>> len(by) ⑤
6
>>> by[0] ⑥
97
>>> by[0] = 102 ⑦

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

1. Per definire un oggetto bytes, usate la sintassi `b''` per i “letterali byte”. Ogni byte nel letterale byte può essere un carattere ASCII o un numero esadecimale codificato da `\x00` a `\xff` (0–255).
2. Il tipo di un oggetto bytes è `bytes`.
3. Esattamente come con le liste e le stringhe, potete ottenere la lunghezza di un oggetto bytes con la funzione built-in `len()`.
4. Esattamente come con le liste e le stringhe, potete usare l'operatore `+` per concatenare oggetti bytes. Il risultato è un nuovo oggetto bytes.
5. Concatenare un oggetto bytes di 5 byte e un oggetto bytes di 1 byte dà come risultato un oggetto bytes di 6 byte.

6. Esattamente come con le liste e le stringhe, potete usare la notazione a indici per ottenere i singoli byte in un oggetto bytes. Gli elementi di una stringa sono stringhe; gli elementi di un oggetto bytes sono interi. Nello specifico, interi tra 0 e 255.
7. Un oggetto bytes è immutabile, quindi non potete assegnare singoli byte. Se avete bisogno di cambiare i singoli byte, potete usare i metodi per affettare le stringhe e gli operatori di concatenazione (che funzionano allo stesso modo delle stringhe), oppure potete convertire l'oggetto bytes in un oggetto bytearray.

```
>>> by = b'abcd\x65'
>>> barr = bytearray(by) ①
>>> barr
bytearray(b'abcde')
>>> len(barr) ②
5
>>> barr[0] = 102 ③
>>> barr
bytearray(b'fbcd')
```

1. Per convertire un oggetto bytes in un oggetto bytearray modificabile, usate la funzione built-in bytearray().
2. Tutti i metodi e le operazioni che si possono eseguire su un oggetto bytes si possono eseguire anche su un oggetto bytearray.
3. L'unica differenza è che, con un oggetto bytearray, potete assegnare i singoli byte utilizzando la notazione a indici. Il valore assegnato deve essere un intero tra 0 e 255.

L'unica cosa che *non potete mai fare* è mescolare stringhe e byte.

```

>>> by = b'd'
>>> s = 'abcde'
>>> by + s                                ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
>>> s.count(by)                            ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly
>>> s.count(by.decode('ascii'))           ③
1

```

1. Non potete concatenare stringhe e byte. Sono due tipi di dato differenti.
2. Non potete contare le occorrenze di byte in una stringa, perché non c'è alcun byte in una stringa. Una stringa è una sequenza di caratteri. Forse intendevate “conta le occorrenze della stringa che si otterrebbe dopo aver decodificato questa sequenza di byte in una particolare codifica di caratteri”? Ebbene, allora dovete dirlo esplicitamente. Python 3 non convertirà implicitamente byte in stringhe o stringhe in byte.
3. Per una sorprendente coincidenza, questa riga di codice dice “conta le occorrenze della stringa che si otterrebbe dopo aver decodificato questa sequenza di byte in una particolare codifica di caratteri”.

Ed ecco il collegamento tra stringhe e byte: gli oggetti bytes hanno un metodo `decode()` che prende una codifica di caratteri e restituisce una stringa, e le stringhe hanno un metodo `encode()` che prende una codifica di caratteri e restituisce un oggetto bytes. Nell'esempio precedente, la decodifica era relativamente semplice — convertire una sequenza di byte in una stringa di caratteri attraverso la codifica ASCII. Ma lo stesso procedimento funziona con qualsiasi codifica che supporti i caratteri della stringa — persino codifiche legacy (non Unicode).

```

>>> a_string = '深入 Python' ①
>>> len(a_string)
9
>>> by = a_string.encode('utf-8') ②
>>> by
b'\xe6\xb7\xb1\xe5\x85\xa5 Python'
>>> len(by)
13
>>> by = a_string.encode('gb18030') ③
>>> by
b'\xc9\xee\xc8\xeb Python'
>>> len(by)
11
>>> by = a_string.encode('big5') ④
>>> by
b'\xb2`\xa4J Python'
>>> len(by)
11
>>> roundtrip = by.decode('big5') ⑤
>>> roundtrip
'深入 Python'
>>> a_string == roundtrip
True


```

1. Questa è una stringa. Ha nove caratteri.
2. Questo è un oggetto bytes. Ha 13 byte. È la sequenza di byte che si ottiene codificando a_string in UTF-8.
3. Questo è un oggetto bytes. Ha 11 byte. È la sequenza di byte che si ottiene codificando a_string in GB18030.
4. Questo è un oggetto bytes. Ha 11 byte. È una *sequenza completamente diversa di byte* che si ottiene codificando a_string in BIG5.
5. Questa è una stringa. Ha nove caratteri. È la sequenza di caratteri che si ottiene decodificando by con l'algoritmo di codifica BIG5. È identica alla stringa originale.



4.7. POST SCRIPTUM: CODIFICA DI CARATTERI PER IL CODICE SORGENTE PYTHON

Python 3 assume che il vostro codice sorgente — cioè ogni file `.py` — sia codificato in UTF-8.

 In Python 2, la codifica di default per i file `.py` era ASCII. In Python 3, la codifica di default è UTF-8.

Se volete usare una codifica differente nel vostro codice Python, potete inserire una dichiarazione di codifica nella prima riga di ogni file. Questa dichiarazione definisce `WINDOWS-1252` come codifica di un file `.py`

```
# -*- coding: windows-1252 -*-
```

Tecnicamente, la dichiarazione per la nuova codifica di carattere si può trovare anche sulla seconda riga, se la prima riga contiene un comando hash-bang di tipo UNIX.

```
#!/usr/bin/python3
# -*- coding: windows-1252 -*-
```

Per maggiori informazioni, consultate la PEP 263: Definire le codifiche per il codice sorgente Python.



4.8. LETTURE DI APPROFONDIMENTO

Su Unicode in Python:

- [Guida pratica a Unicode e Python](#)
- [Cosa c'è di nuovo in Python 3: testo vs. dati invece di Unicode vs. 8-bit](#) [La PEP 261](#) spiega il modo in cui Python gestisce i caratteri astrali al di fuori del Piano Multilinguistico di Base (in inglese, Basic Multilingual Plane), cioè i caratteri il cui valore ordinale è maggiore di 65535.

Su Unicode in generale:

- [Il minimo sindacale che ogni sviluppatore di software deve assolutamente sapere su Unicode e sugli insiemi di caratteri \(niente scuse!\)](#)
- [L'essenza di Unicode](#)
- [Le stringhe di caratteri](#)
- [Caratteri vs. byte](#)

Sulle codifiche di carattere in altri formati:

- [La codifica di carattere in XML](#)
- [La codifica di carattere in HTML](#)

Sulle stringhe e sulla formattazione di stringhe:

- [string — Operazioni comuni sulle stringhe](#)
- [La sintassi della formattazione di stringhe](#)
- [Mini-linguaggio per le specifiche di formato](#)
- [PEP 3101: Formattazione avanzata delle stringhe](#)

CAPITOLO 5. ESPRESSIONI REGOLARI

“Alcune persone, quando affrontano un problema, pensano: ‘Ho capito, userò le espressioni regolari.’ Ora hanno due problemi.”

— Jamie Zawinski

5.1. IMMERSIONE!

Generalmente, estrarre una specifica porzione da un testo molto lungo non è un’impresa facile. In Python le stringhe sono dotate di metodi per effettuare ricerche e sostituzioni: `index()`, `find()`, `split()`, `count()`, `replace()`, &c. Ma questi metodi si limitano a gestire i casi più semplici. Per esempio, il metodo `index()` cerca una singola sottostringa costante, e la ricerca è sempre sensibile alle maiuscole. Per fare ricerche insensibili alle maiuscole di una stringa `s`, dovete chiamare `s.lower()` oppure `s.upper()` e assicurarvi che le vostre stringhe di ricerca contengano i caratteri appropriati per corrispondere. I metodi `replace()` e `split()` hanno le stesse limitazioni.

Se il vostro obiettivo può essere raggiunto con i metodi delle stringhe, dovrete usarli. Sono veloci, semplici e facili da leggere, e c’è molto da dire a favore del codice veloce, semplice e leggibile. Ma se state lavorando con le stringhe e utilizzate una grande quantità di funzioni differenti insieme a una serie di istruzioni `if` per gestire casi particolari, o se state combinando tra loro chiamate a `split()` e `join()` per affettare e ricomporre le vostre stringhe, potreste aver bisogno di avvalervi delle espressioni regolari.

Le espressioni regolari sono un modo potente e (per la maggior parte) standard per cercare, sostituire e riconoscere testo tramite complessi pattern di caratteri. Sebbene la sintassi delle espressioni regolari sia ermetica e diversa dal normale codice, il risultato potrebbe rivelarsi *più* leggibile di una soluzione manuale che usa una lunga catena di metodi delle stringhe. Le espressioni regolari vi permettono anche di inserire commenti al loro interno, in modo che possiate documentare le singole parti che le compongono.



Se avete usato le espressioni regolari in altri linguaggi (come Perl, JavaScript, o PHP), la sintassi utilizzata da Python vi sarà molto familiare. Leggete il riepilogo della documentazione ufficiale sul modulo re per avere una descrizione delle funzioni disponibili e dei loro argomenti.



5.2. CASO DI STUDIO: INDIRIZZI DELLE VIE

Questa serie di esempi si ispira a un problema reale incontrato sul lavoro diversi anni fa, quando ho avuto bisogno di ripulire e uniformare gli indirizzi delle vie estratti da un sistema legacy prima di inserirli in un sistema più nuovo. (Vedete, non mi invento questa roba sul momento; è veramente utile.) Questo esempio mostra come ho inizialmente affrontato il problema.

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.') ①
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.') ②
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.') ③
'100 NORTH BROAD RD.'
>>> import re ④
>>> re.sub('ROAD$', 'RD.', s) ⑤
'100 NORTH BROAD RD.'
```

- I. Il mio obiettivo è uniformare un indirizzo in modo che 'ROAD' sia sempre abbreviato come 'RD.'. A prima vista, pensavo fosse una cosa abbastanza semplice da non dover usare altro che il metodo `replace()` delle stringhe. Dopo tutto, i dati erano già in maiuscolo, così eventuali corrispondenze mancate con caratteri minuscoli non sarebbero state un problema. E la stringa di ricerca, 'ROAD', era una costante. E in questo caso ingannevolmente semplice, `s.replace()` effettivamente funziona.

2. La vita, sfortunatamente, è piena di controesempi, e ne ho subito trovato uno. Il problema qui è che 'ROAD' compare due volte nell'indirizzo, una volta come parte del nome della via 'BROAD' e una volta come parola a sé. Il metodo `replace()` vede queste due occorrenze e le sostituisce entrambe senza distinzioni; nel frattempo, io vedo i miei indirizzi che vengono distrutti.
3. Per risolvere il problema degli indirizzi che contengono più di una sottostringa 'ROAD', potreste ricorrere a qualcosa di simile a questo: cercate e sostituite 'ROAD' solo negli ultimi quattro caratteri dell'indirizzo (`s[-4:]`) e lasciate stare il resto della stringa (`s[:-4]`). Ma, come potete vedere, questa strategia rischia di diventare immediatamente poco pratica. Per esempio, il pattern dipende dalla lunghezza della stringa che state sostituendo. (Se doveste sostituire 'STREET' con 'ST.', doveste usare `s[:-6]` e `s[-6:].replace(...)`.) Vorreste tornare indietro dopo sei mesi e correggere questo dettaglio? Io so che non vorrei.
4. È il momento di avvalersi delle espressioni regolari. In Python, tutte le funzionalità relative alle espressioni regolari sono contenute nel modulo `re`.
5. Date un'occhiata al primo parametro: 'ROAD\$'. Questa è una semplice espressione regolare che corrisponde a 'ROAD' solo quando si trova alla fine di una stringa. Il simbolo `$` significa "fine della stringa". (C'è un carattere corrispondente, il caret `^`, che significa "inizio della stringa".) Usando la funzione `re.sub()`, cercate l'espressione regolare 'ROAD\$' nella stringa `s` e la sostituite con 'RD.'. La funzione trova una corrispondenza con ROAD alla fine della stringa `s`, ma *non* trova alcuna corrispondenza con la ROAD che è parte della parola BROAD, perché quella è nel mezzo di `s`.

Proseguendo il mio racconto sulla ripulitura degli indirizzi, ho presto scoperto che la strategia precedente, cioè trovare una corrispondenza con 'ROAD' alla fine dell'indirizzo, non era poi così adeguata, perché non tutti gli indirizzi includono una designazione della via. Alcuni indirizzi terminano semplicemente con il nome della via. Riuscivo a cavarmela per la maggior parte del tempo, ma se il nome della via fosse stato 'BROAD', allora l'espressione regolare avrebbe trovato una corrispondenza di 'ROAD' alla fine della stringa come parte della parola 'BROAD', ma questo non era ciò che volevo.

*^ corrisponde
all'inizio di
una stringa.
\$ corrisponde*


```

>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s) ①
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s) ②
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s) ③
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) ④
'100 BROAD RD. APT 3'

```

*alla fine di
una stringa.*

1. Quello che *realmente* volevo era trovare una corrispondenza con 'ROAD' quando era al termine della stringa ed era una parola a sé (non parte di una parola più grande). Le espressioni regolari usano `\b` per esprimere che “il limite di una parola deve trovarsi proprio qui”. In Python, l’uso di questo pattern è complicato dalla necessità di effettuare l’escape di ogni carattere di backslash `\` contenuto in una stringa. Questa seccatura viene solitamente chiamata la piaga del backslash, ed è una delle ragioni per cui le espressioni regolari sono più semplici da usare in Perl che in Python. Sfortunatamente, Perl mescola le espressioni regolari con altra sintassi, quindi se avete un problema potrebbe essere difficile capire se è un bug nella sintassi o è un bug nella vostra espressione regolare.
2. Per evitare la piaga del backslash, potete usare quella che viene chiamata *stringa raw*, aggiungendo alla stringa un prefisso con la lettera `r`. Questo prefisso indica che non deve essere eseguito alcun escape sul contenuto di quella stringa; `'\t'` è un carattere di tabulazione, ma `r'\t'` è effettivamente il carattere di backslash `\` seguito dalla lettera `t`. Vi raccomando di usare sempre stringhe raw quando avete a che fare con espressioni regolari, altrimenti le cose diventano troppo confuse troppo velocemente (e le espressioni regolari sono già abbastanza confuse per conto proprio).
3. **sigh** Sfortunatamente, ho subito trovato altri esempi che contraddicevano la mia logica. In questo caso, l’indirizzo conteneva la parola 'ROAD' come parola intera, ma non era alla fine, perché l’indirizzo aveva un numero di appartamento dopo il nome della via. Dato che 'ROAD' non si trova alla fine della stringa, non c’è alcuna corrispondenza, quindi l’intera chiamata a `re.sub()` non riesce a sostituire nulla e restituisce come risultato la stringa originale, ma questo non è ciò che desiderate.

4. Per risolvere questo problema, ho rimosso il carattere \$ e ho aggiunto un altro \b. Ora l'espressione regolare dice “trova una corrispondenza con 'ROAD' quando è una parola intera in qualsiasi punto della stringa”, che sia alla fine, all'inizio, o da qualche parte nel mezzo.



5.3. CASO DI STUDIO: NUMERI ROMANI

Avrete probabilmente già incontrato i numeri romani, anche se non li avete riconosciuti. Potreste averli visti nei copyright di vecchi film e spettacoli televisivi (“Copyright MCMXLVI” invece di “Copyright 1946”) oppure nelle epigrafi sui muri di biblioteche o università (“fondata nell'anno MDCCCLXXXVIII” invece di “fondata nel 1888”). Potreste averli visti anche negli indici dei libri o nei riferimenti bibliografici. È un sistema di rappresentazione numerica che risale all'antico Impero romano (da cui il nome).

Nei numeri romani, ci sono sette caratteri che sono ripetuti e combinati in vari modi per rappresentare i numeri.

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

Le seguenti sono alcune regole generali per costruire i numeri romani.

- Talvolta i caratteri sono additivi. I è 1, II è 2, e III è 3. VI è 6 (letteralmente, “5 e 1”), VII è 7, e VIII è 8.
- I caratteri delle potenze di dieci (I, X, C e M) possono essere ripetuti fino a tre volte. A 4, dovete sottrarre dal carattere del quintuplo più alto successivo. Non potete rappresentare 4 come IIII; invece, va rappresentato come IV (“1 meno di 5”). Il numero 40 è scritto come XL (“10 meno di 50”), 41 come XLI, 42 come XLII, 43 come XLIII, e poi 44 come XLIV (“10 meno di 50, poi 1 meno di 5”).

- Talvolta i caratteri sono... l'opposto di additivi. Mettendo certi caratteri prima di altri, sottraete dal valore finale. Per esempio, a 9, dovete sottrarre dal carattere della potenza di dieci più alta successiva: 8 è VIII, ma 9 è IX ("1 meno di 10"), non VIIII (dato che il carattere I non può essere ripetuto quattro volte). Il numero 90 è XC, 900 è CM.
- I caratteri dei quintupli non possono essere ripetuti. Il numero 10 è sempre rappresentato come X, mai come vv. Il numero 100 è sempre C, mai LL.
- I numeri romani vengono sempre letti da sinistra a destra, così l'ordine dei caratteri ha molta importanza. DC è 600; CD è un numero completamente differente (400, "100 meno di 500"). CI è 101; IC non è nemmeno un numero romano valido (perché non potete sottrarre 1 direttamente da 100; dovrete scriverlo come XCIX, cioè "10 meno di 100, poi 1 meno di 10").

5.3.1. CONTROLLARE LE MIGLIAIA

Come potremmo fare per verificare che una stringa arbitraria sia un numero romano valido? Prendiamo una cifra alla volta. Dato che i numeri romani sono sempre scritti dalla cifra più alta a quella più bassa, cominciamo con la più alta: le migliaia. Per i numeri da 1000 in su, le migliaia sono rappresentate da una serie di caratteri M.

```
>>> import re
>>> pattern = '^M?M?M?$'          ①
>>> re.search(pattern, 'M')        ②
<_sre.SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')       ③
<_sre.SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')      ④
<_sre.SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM')     ⑤
>>> re.search(pattern, '')          ⑥
<_sre.SRE_Match object at 0106F4A8>
```

1. Questo pattern è composto da tre parti. ^ fa corrispondere quanto segue solo se è all'inizio della stringa. Se questo non fosse specificato, il pattern troverebbe una corrispondenza a prescindere dalla posizione dei caratteri M, che non è ciò che volete. Voi volete essere sicuri che i caratteri M, se ce ne sono, siano all'inizio della stringa. M? rappresenta una corrispondenza opzionale con un singolo carattere M. Ripetuto tre volte,

rappresenta una corrispondenza con una stringa che contiene da zero a tre caratteri `M` di seguito. E `$` corrisponde alla fine della stringa. Combinato con il carattere `^` all'inizio, questo significa che il pattern cerca una corrispondenza con l'intera stringa, senza nessun altro carattere prima o dopo i caratteri `M`.

2. L'essenza del modulo `re` è la funzione `search()`, che prende un'espressione regolare (pattern) e una stringa ('`M`') per provare a trovarvi una corrispondenza con l'espressione regolare. Se viene trovata una corrispondenza, `search()` restituisce un oggetto che ha vari metodi per descriverla; se non c'è alcuna corrispondenza, `search()` restituisce `None`, il valore nullo di Python. Tutto quello che vi interessa al momento è se il pattern trova una corrispondenza, cosa che potete capire guardando solamente al valore restituito da `search()`. '`M`' corrisponde a questa espressione regolare, perché la prima `M` opzionale corrisponde e la seconda e la terza `M` opzionali vengono ignorate.
3. '`MM`' corrisponde perché la prima e la seconda `M` opzionali corrispondono e la terza `M` viene ignorata.
4. '`MMM`' corrisponde perché tutti e tre i caratteri `M` corrispondono.
5. '`MMMM`' non corrisponde. Tutti e tre i caratteri `M` corrispondono, poi l'espressione regolare insiste sulla fine della stringa (a causa del carattere `$`) ma la stringa non è ancora terminata (a causa della quarta `M`). Quindi `search()` restituisce `None`.
6. È interessante notare che una stringa vuota corrisponde a questa espressione regolare perché tutti i caratteri `M` sono opzionali.

5.3.2. CONTROLLARE LE CENTINAIA

Le centinaia sono più difficili da trattare rispetto alle migliaia perché ci sono diversi modi mutuamente esclusivi in cui possono essere espresse a seconda del loro valore.

- 100 = C
- 200 = CC
- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

*? rende un
pattern
opzionale.*

Quindi ci sono quattro possibili pattern:

- CM;
- CD;
- da zero a tre caratteri C (zero se la cifra delle centinaia nel numero intero corrispondente è 0);
- D, seguito da zero fino a tre caratteri C.

Gli ultimi due pattern possono essere combinati nel modo seguente:

- una D opzionale, seguita da zero fino a tre caratteri C.

Questo esempio mostra come validare le centinaia di un numero romano.

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$' ①
>>> re.search(pattern, 'MCM') ②
<_sre.SRE_Match object at 01070390>
>>> re.search(pattern, 'MD') ③
<_sre.SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC') ④
<_sre.SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC') ⑤
>>> re.search(pattern, '') ⑥
<_sre.SRE_Match object at 01071D98>
```

1. Questo pattern comincia allo stesso modo del precedente, controllando l'inizio della stringa (^), poi le migliaia (M?M?M?). Dopo prosegue con la nuova parte, tra parentesi, che definisce un insieme di tre pattern mutuamente esclusivi, separati da barre verticali: CM, CD e D?C?C?C? (che contiene una D opzionale seguita da zero fino a tre caratteri C opzionali). Il riconoscitore di espressioni regolari controlla ognuno di questi pattern nell'ordine dato (da sinistra a destra), prende il primo che corrisponde e ignora il resto.
2. 'MCM' corrisponde perché la prima M corrisponde, il secondo e il terzo carattere M vengono ignorati e CM corrisponde (quindi i pattern CD e D?C?C?C? non vengono mai considerati). MCM è la rappresentazione di 1900 come numero romano.

3. 'MD' corrisponde perché la prima M corrisponde, il secondo e il terzo carattere M vengono ignorati e il pattern `D?C?C?C?` corrisponde a D (ognuno dei tre caratteri C opzionali viene ignorato). MD è la rappresentazione di 1500 come numero romano.
4. 'MMMCCC' corrisponde perché tutti e tre i caratteri M corrispondono e il pattern `D?C?C?C?` corrisponde a CCC (la D è opzionale e viene ignorata). MMMCCC è la rappresentazione di 3300 come numero romano.
5. 'MCMC' non corrisponde. La prima M corrisponde, il secondo e il terzo carattere M vengono ignorati e CM corrisponde, ma poi `$` non corrisponde perché non siete ancora alla fine della stringa (avete ancora un carattere C senza corrispondenza). Il carattere C non corrisponde come parte del pattern `D?C?C?C?` perché il pattern mutuamente esclusivo CM ha già trovato una corrispondenza.
6. È interessante notare che una stringa vuota corrisponde ancora a questo pattern, perché tutti i caratteri M sono opzionali e vengono ignorati e la stringa vuota corrisponde al pattern `D?C?C?C?` dove tutti i caratteri sono opzionali e vengono ignorati.

Whew! Vedete come le espressioni regolari possono diventare brutte velocemente? E avete solo trattato le migliaia e le centinaia dei numeri romani. Se però siete riusciti a seguire tutto questo, le decine e le unità sono facili, perché seguono esattamente lo stesso schema. Ma prima diamo un'occhiata a un altro modo di esprimere il pattern.



5.4. USARE LA SINTASSI {n,m}

Nella sezione precedente, avevate a che fare con un pattern dove lo stesso carattere poteva essere ripetuto fino a tre volte. C'è un altro modo, che alcune persone trovano più leggibile, per esprimere questo pattern in un'espressione regolare. Prima di tutto diamo un'occhiata al metodo che abbiamo già usato nell'esempio precedente.

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')      ①
<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM')     ②
<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM')    ③
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM')   ④
>>>
```

{1,4}
corrisponde
da 1 fino a 4
occorrenze di
un pattern.

1. Questo trova una corrispondenza con l'inizio della stringa, poi con la prima M opzionale ma non con la seconda e la terza M (ma questo è OK perché sono opzionali), e poi con la fine della stringa.
2. Questo trova una corrispondenza con l'inizio della stringa, poi con la prima e la seconda M opzionali ma non con la terza M (ma questo è OK perché è opzionale), e poi con la fine della stringa.
3. Questo trova una corrispondenza con l'inizio della stringa, poi con tutte e tre le M opzionali, e poi con la fine della stringa.
4. Questo trova una corrispondenza con l'inizio della stringa, poi con tutte e tre le M opzionali, ma non con la fine della stringa (perché c'è ancora una M senza corrispondenza), quindi il pattern non corrisponde e la funzione restituisce None.

```

>>> pattern = '^M{0,3}$'           ①
>>> re.search(pattern, 'M')         ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM')        ③
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM')        ④
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM')       ⑤
>>>

```

1. Questo pattern dice: “Trova una corrispondenza con l’inizio della stringa, poi da zero a tre caratteri M, poi la fine della stringa.” Lo 0 e il 3 possono essere qualsiasi numero; se volete trovare una corrispondenza con almeno uno ma non più di tre caratteri M, potete usare `M{1,3}`.
2. Questo trova una corrispondenza con l’inizio della stringa, poi una di tre possibili M, poi la fine della stringa.
3. Questo trova una corrispondenza con l’inizio della stringa, poi due di tre possibili M, poi la fine della stringa.
4. Questo trova una corrispondenza con l’inizio della stringa, poi tre di tre possibili M, poi la fine della stringa.
5. Questo trova una corrispondenza con l’inizio della stringa, poi tre di tre possibili M, ma *non* con la fine della stringa. L’espressione regolare permette solo fino a tre caratteri M prima della fine della stringa, ma ne avete quattro, quindi il pattern non corrisponde e la funzione restituisce `None`.

5.4.1. CONTROLLARE LE DECINE E LE UNITÀ

Ora espandiamo l’espressione regolare dei numeri romani per trattare le decine e le unità. Questo esempio mostra il controllo sulle decine.


```

>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')      ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')       ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX')      ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')    ④
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX')   ⑤
>>>

```

1. Questo trova una corrispondenza con l'inizio della stringa, poi con la prima M opzionale, poi con CM, poi con XL, poi con la fine della stringa. Ricordate, la sintassi (A|B|C) significa “trova una corrispondenza con esattamente uno solo tra A, B, o C”. Avete trovato una corrispondenza con XL, quindi ignorate le scelte XC e L?X?X?X? e poi vi spostate alla fine della stringa. MCMXL è la rappresentazione di 1940 come numero romano.
2. Questo trova una corrispondenza con l'inizio della stringa, poi con la prima M opzionale, poi con L?X?X?X?. Di L?X?X?X?, trova una corrispondenza con L e tralascia i tre caratteri X opzionali. Poi vi spostate alla fine della stringa. MCML è la rappresentazione di 1950 come numero romano.
3. Questo trova una corrispondenza con l'inizio della stringa, poi con la prima M opzionale, poi con CM, poi con la L opzionale e con la prima X opzionale, tralascia la seconda e la terza X opzionali, poi arriva alla fine della stringa. MCMLX è la rappresentazione di 1960 come numero romano.
4. Questo trova una corrispondenza con l'inizio della stringa, poi con la prima M opzionale, poi con CM, poi con la L opzionale e con tutti e tre i caratteri X opzionali, poi con la fine della stringa. MCMLXXX è la rappresentazione di 1980 come numero romano.
5. Questo trova una corrispondenza con l'inizio della stringa, poi con la prima M opzionale, poi con CM, poi con la L opzionale e con tutti e tre i caratteri X opzionali, ma poi *fallisce la corrispondenza* con la fine della stringa perché c'è ancora una X di cui dar conto. Quindi l'intero pattern non corrisponde e la funzione restituisce None. MCMLXXXX non è un numero romano valido.

L'espressione per le unità segue lo stesso schema. Vi risparmierei i dettagli e vi mostrerò il risultato finale.

$(A | B)$

*corrisponde al
pattern A
oppure al
pattern B, ma
non a
entrambi.*

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

Come apparirebbe se usassimo la sintassi sostitutiva $\{n,m\}$? Questo esempio mostra la nuova sintassi.

```
>>> pattern = '^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV') ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI') ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCLXXXVIII') ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I') ④
<_sre.SRE_Match object at 0x008EEB48>
```

1. Questo trova una corrispondenza con l'inizio della stringa, poi con uno di tre possibili caratteri M, poi con $D?C\{0,3\}$. Di questo, trova una corrispondenza con la D opzionale e con zero di tre possibili caratteri C. Proseguendo, trova una corrispondenza con $L?X\{0,3\}$ tramite la L opzionale e zero di tre possibili caratteri X. Poi trova una corrispondenza con $V?I\{0,3\}$ tramite la V opzionale e zero di tre possibili caratteri I, e da ultimo con la fine della stringa. MDLV è la rappresentazione di 1555 come numero romano.
2. Questo trova una corrispondenza con l'inizio della stringa, poi con due di quattro possibili caratteri M, poi con $D?C\{0,3\}$ tramite una D e uno di tre possibili caratteri C; poi con $L?X\{0,3\}$ tramite una L e uno di tre possibili caratteri X; poi con $V?I\{0,3\}$ tramite una V e uno di tre possibili caratteri I; poi con la fine della stringa. MMDCLXVI è la rappresentazione di 2666 come numero romano.
3. Questo trova una corrispondenza con l'inizio della stringa, poi con tre su tre caratteri M, poi con $D?C\{0,3\}$ tramite una D e tre su tre caratteri C; poi con $L?X\{0,3\}$ tramite una L e tre su tre caratteri X; poi con $V?I\{0,3\}$ tramite una V e tre su tre caratteri I; poi con la fine della stringa. MMMDCCCLXXXVIII è la rappresentazione di 3888 come numero romano, ed è il numero romano più lungo che potete scrivere senza la sintassi estesa.
4. Guardate attentamente. (Mi sento come un mago: “Guardate attentamente, bambini. Sto per estrarre un coniglio dal mio cappello.”) Questo trova una corrispondenza con l'inizio della stringa, poi con zero su tre M, poi trova una corrispondenza con $D?C\{0,3\}$ tralasciando la D opzionale e trovando zero su tre C, poi trova una corrispondenza con $L?X\{0,3\}$ tralasciando la L opzionale e trovando zero su tre X, poi trova una corrispondenza con $V?I\{0,3\}$ tralasciando la V opzionale e trovando una su tre I. Poi trova la fine della stringa. Whoa.

Se avete seguito tutto questo e lo avete capito alla prima lettura, state andando meglio di me. Ora immaginate di provare a capire l'espressione regolare di qualcun altro in mezzo a una funzione critica di un lungo programma. O immaginate addirittura di tornare a una vostra espressione regolare qualche mese dopo averla scritta. Io l'ho fatto, e non è stata una bella esperienza.

Ora esploriamo una sintassi alternativa che può aiutarvi a mantenere le vostre espressioni regolari.



5.5. ESPRESSIONI REGOLARI VERBOSE

Finora avete avuto a che fare solamente con quelle che chiamerò espressioni regolari “compatte”. Come avete visto, sono difficili da leggere, e anche se capite cosa fa una di esse non c’è alcuna garanzia che sarete in grado di capirlo nuovamente sei mesi dopo. Quello di cui avete davvero bisogno è di inserire documentazione in linea.

Python vi permette di farlo grazie a quelle che vengono chiamate *espressioni regolari verbose*. Un’espressione regolare verbosa è diversa da un’espressione regolare compatta sotto due aspetti.

- Lo spazio bianco viene ignorato. Spazi, tabulazioni e ritorni a capo non corrispondono a spazi, tabulazioni e ritorni a capo. Non vengono mai considerati. (Se volete utilizzare uno spazio in un’espressione regolare verbosa, dovete effettuarne l’escape mettendogli davanti un backslash.)
- I commenti sono ignorati. Un commento in un’espressione regolare verbosa è come un commento nel codice Python: comincia con un carattere # e prosegue fino alla fine della riga. In questo caso è un commento all’interno di una stringa su più righe invece che all’interno del vostro codice sorgente, ma funziona allo stesso modo.

Queste differenze diventeranno più chiare con un esempio. Rivediamo l’espressione regolare compatta con la quale stavate lavorando e trasformiamola in un’espressione regolare verbosa. Questo esempio mostra come fare.

```

>>> pattern = '''
^                # inizio della stringa
M{0,3}           # migliaia - da 0 a 3 M
(CM|CD|D?C{0,3}) # centinaia - 900 (CM), 400 (CD), 0-300 (da 0 a 3 C),
                  # o 500-800 (D, seguita da 0 fino a 3 C)
(XC|XL|L?X{0,3}) # decine - 90 (XC), 40 (XL), 0-30 (da 0 a 3 X),
                  # o 50-80 (L, seguita da 0 fino a 3 X)
(IX|IV|V?I{0,3}) # unità - 9 (IX), 4 (IV), 0-3 (da 0 a 3 I),
                  # o 5-8 (V, seguita da 0 fino a 3 I)
$                # fine della stringa
'''

>>> re.search(pattern, 'M', re.VERBOSE)           ①
<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)  ②
<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'MMMDCCLXXXVIII', re.VERBOSE) ③
<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'M')                        ④

```

1. La cosa più importante da ricordare quando usate le espressioni regolari verbose è che avete bisogno di passare un argomento aggiuntivo quando lavorate con esse: `re.VERBOSE` è una costante definita nel modulo `re` usata per segnalare che il pattern dovrebbe essere trattato come un'espressione regolare verbosa. Come potete vedere, questo pattern ha un bel po' di spazi bianchi (che vengono tutti ignorati) e diversi commenti (che vengono tutti ignorati). Una volta che ignorate gli spazi bianchi e i commenti, questa è esattamente la stessa espressione regolare che avete visto nella sezione precedente, ma è molto più leggibile.
2. Questo trova una corrispondenza con l'inizio della stringa, poi con una di tre possibili M, poi con CM, poi con L e con tre di tre possibili X, poi con IX, poi con la fine della stringa.
3. Questo trova una corrispondenza con l'inizio della stringa, poi con tre di tre possibili M, poi con D e con tre di tre possibili C, poi con L e con tre di tre possibili X, poi con V e con tre di tre possibili I, poi con la fine della stringa.
4. Questo non trova alcuna corrispondenza. Perché? Perché non ha il flag `re.VERBOSE`, quindi la funzione `re.search()` tratta il pattern come un'espressione regolare compatta, dove gli spazi bianchi hanno significato e i simboli di hash sono letterali. Python non è in grado di scoprire da solo se un'espressione regolare è

verbosa oppure no. Python presume che ogni espressione regolare sia compatta a meno che non gli diciate esplicitamente che è verbosa.



5.6. CASO DI STUDIO: RICONOSCERE I NUMERI DI TELEFONO

Finora vi siete concentrati nel trovare corrispondenze con interi pattern. Un pattern corrisponde, oppure no. Ma le espressioni regolari sono molto più potenti di così. Quando un'espressione regolare *trova* una corrispondenza, potete recuperarne delle parti specifiche. Potete trovare cosa corrisponde dove.

Questo esempio proviene da un altro problema reale, incontrato ancora una volta in un precedente lavoro. Il problema: riconoscere un numero telefonico americano. Il cliente voleva essere in grado di introdurre il numero senza alcun vincolo (in un singolo campo), ma poi voleva memorizzare separatamente nel database aziendale il codice d'area, il prefisso, il numero e opzionalmente un'estensione. Ho rovistato nel Web e trovato molti esempi di espressioni regolari che dichiaravano di fare proprio questo, ma nessuna era abbastanza permissiva.

Ecco i numeri telefonici che dovevo essere in grado di accettare:

- 800-555-1212
- 800 555 1212
- 800.555.1212
- (800) 555-1212

\d
corrisponde a
qualsiasi cifra
numerica
(0-9). \D
corrisponde a
qualsiasi cosa
tranne le
cifre.

- 1-800-555-1212
- 800-555-1212-1234
- 800-555-1212x1234
- 800-555-1212 ext. 1234
- work 1-(800) 555.1212 #1234

Una gran varietà! In ognuno di questi casi, avevo bisogno di sapere che il codice d'area era 800, il prefisso era 555 e il resto del numero telefonico era 1212. Per quelli con un'estensione, avevo bisogno di sapere che quella estensione era 1234.

Lavoriamo per sviluppare una soluzione al riconoscimento dei numeri di telefono. Questo esempio mostra il primo passo.

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$') ①
>>> phonePattern.search('800-555-1212').groups()           ②
('800', '555', '1212')
>>> phonePattern.search('800-555-1212-1234')               ③
>>> phonePattern.search('800-555-1212-1234').groups()       ④

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```

1. Leggete sempre un'espressione regolare da sinistra a destra. Questa trova una corrispondenza con l'inizio della stringa e poi con `(\d{3})`. Cos'è `\d{3}`? Be', `\d` significa "qualsiasi cifra numerica" (da 0 fino a 9). E `{3}` significa "trova una corrispondenza con esattamente tre cifre numeriche"; è una variazione della sintassi `{n,m}` che avete visto prima. Mettere il tutto tra parentesi significa "trova una corrispondenza con esattamente tre cifre numeriche e *memorizzala come un gruppo che posso richiederti più tardi*". Poi l'espressione regolare trova una corrispondenza con un trattino letterale. Poi trova una corrispondenza con un altro gruppo di esattamente tre cifre. Poi con un altro trattino letterale. Poi con un altro gruppo di esattamente quattro cifre. Poi trova una corrispondenza con la fine della stringa.
2. Per avere accesso ai gruppi che il motore di espressioni regolari ha memorizzato durante il riconoscimento, usate il metodo `groups()` sull'oggetto che il metodo `search()` vi restituisce. Invocare `groups()` vi restituirà una tupla di tanti gruppi quanti ne avete definiti nell'espressione regolare. In questo caso, avete definito tre gruppi: uno di tre cifre, uno di tre cifre e uno di quattro cifre.

3. Questa espressione regolare non è la risposta finale, perché non gestisce un numero telefonico con un'estensione alla fine. Per fare questo, avrete bisogno di espandere l'espressione regolare.
4. E questo è il motivo per cui non dovrete mai “concatenare” i metodi `search()` e `group()` nel codice di produzione. Se il metodo `search()` non restituisce alcuna corrispondenza, restituisce `None` invece di un oggetto corrispondenza delle espressioni regolari. Ovviamente, invocare `None.groups()` solleva un'eccezione: `None` non possiede un metodo `groups()`. (Naturalmente, la cosa è un po' meno ovvia quando vedete sorgere questa eccezione dalle profondità del vostro codice. Sì, parlo per esperienza qui.)

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$') ①
>>> phonePattern.search('800-555-1212-1234').groups()           ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')                     ③
>>>
>>> phonePattern.search('800-555-1212')                           ④
>>>
```

1. Questa espressione regolare è quasi identica alla precedente. Esattamente come prima, trova una corrispondenza con l'inizio della stringa, poi con un gruppo di tre cifre da memorizzare, poi con un trattino, poi con un gruppo di tre cifre da memorizzare, poi con un trattino, poi con un gruppo di quattro cifre da memorizzare. La novità è che poi trova una corrispondenza con un altro trattino, poi con un gruppo di una o più cifre da memorizzare, e poi con la fine della stringa.
2. Il metodo `groups()` ora restituisce una tupla di quattro elementi, dato che l'espressione regolare ora definisce quattro gruppi da memorizzare.
3. Sfortunatamente, anche questa espressione regolare non è la risposta finale, perché presume che le differenti parti di un numero telefonico siano separate da trattini. E se invece fossero separate da spazi, o virgole, o punti? Avete bisogno di una soluzione più generale per trovare una corrispondenza con diversi tipi di separatori.
4. Oops! Non solo questa espressione regolare non fa tutto quello che volete, ma in realtà è un passo indietro, perché ora non potete più riconoscere numeri telefonici *senza* un'estensione. Questo non è per niente quello che volevate; se l'estensione c'è volete sapere qual è, ma se non è presente volete comunque sapere quali sono le diverse parti del numero principale.

L'esempio seguente mostra l'espressione regolare che gestisce i separatori tra le diverse parti del numero di telefono.


```

>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$') ①
>>> phonePattern.search('800 555 1212 1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups() ③
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234') ④
>>>
>>> phonePattern.search('800-555-1212') ⑤
>>>

```

1. Allacciatevi le cinture. State trovando una corrispondenza con l'inizio della stringa, poi con un gruppo di tre cifre, poi con `\D+`. Cosa diavolo è quello? Ebbene, `\D` corrisponde a qualsiasi carattere *tranne* una cifra numerica, e `+` significa “1 o più”. Così `\D+` corrisponde a uno o più caratteri che non sono cifre. Questo è ciò che usate al posto di un trattino letterale per provare a trovare una corrispondenza con separatori differenti.
2. Usare `\D+` invece di `-` significa che ora potete trovare una corrispondenza con numeri le cui parti sono separate da spazi invece che da trattini.
3. Naturalmente, i numeri telefonici separati da trattini funzionano ancora.
4. Sfortunatamente, questa non è ancora la risposta finale, perché presume che ci sia sempre un separatore. E se il numero telefonico fosse introdotto senza nessuno spazio o trattino?
5. Oops! Questo non ha ancora risolto il problema dell'estensione obbligatoria. Ora avete due problemi, ma potete risolverli entrambi con la stessa tecnica.

Il prossimo esempio mostra l'espressione regolare per gestire numeri di telefono *senza* separatori.

```

>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('80055512121234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups() ③
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ④
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234') ⑤
>>>

```

1. L'unico cambiamento che avete fatto rispetto all'ultimo passo è sostituire tutti i + con *. Invece di usare \D+ tra le parti del numero telefonico, ora trovate una corrispondenza con \D*. Ricordate che + significa "l o più"? Ebbene, * significa "zero o più". Così ora dovreste essere in grado di riconoscere i numeri telefonici anche quando non c'è alcun carattere di separazione.
2. E guardate un po', effettivamente funziona. Perché? Trovate una corrispondenza con l'inizio della stringa, poi con un gruppo di tre cifre da memorizzare (800), poi con zero caratteri non numerici, poi con un gruppo di tre cifre da memorizzare (555), poi con zero caratteri non numerici, poi con un gruppo di quattro cifre da memorizzare (1212), poi con zero caratteri non numerici, poi con un gruppo di un numero arbitrario di cifre da memorizzare (1234), poi con la fine della stringa.
3. Anche altre variazioni funzionano, adesso: punti invece di trattini, e sia uno spazio che una x prima dell'estensione.
4. Infine, avete risolto l'altro annoso problema: le estensioni sono di nuovo opzionali. Se nessuna estensione viene trovata, il metodo group() restituisce ancora una tupla di quattro elementi, ma il quarto elemento è solamente una stringa vuota.
5. Odio portare cattive notizie, ma non avete ancora finito. Qual è il problema qui? C'è un carattere in più prima del codice d'area, ma l'espressione regolare presume che il codice d'area sia la prima cosa all'inizio della stringa. Nessun problema, potete usare la stessa tecnica degli "zero o più caratteri non numerici" per saltare i caratteri iniziali prima del codice d'area.

L'esempio seguente mostra come gestire i caratteri iniziali nei numeri di telefono.

```

>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('(800)5551212 ext. 1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ③
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234') ④
>>>

```

1. Questo è lo stesso pattern dell'esempio precedente, a parte il fatto che ora state trovando una corrispondenza con `\D*` (zero o più caratteri non numerici) prima del primo gruppo da memorizzare (il codice d'area). Notate che non state memorizzando questi caratteri non numerici (non sono tra parentesi). Se li trovate, semplicemente li saltate e poi cominciate a memorizzare il codice d'area in qualsiasi punto si trovi.
2. Potete riconoscere con successo anche i numeri telefonici con una parentesi aperta prima del codice d'area. (La parentesi chiusa dopo il codice d'area è già gestita: viene trattata come un separatore non numerico e corrisponde al `\D*` dopo il primo gruppo da memorizzare.)
3. Giusto come prova del nove per assicurarvi di non aver guastato nulla che prima funzionava. Dato che i caratteri iniziali sono interamente opzionali, questo trova una corrispondenza con l'inizio della stringa, poi con zero caratteri non numerici, poi con un gruppo di tre cifre da memorizzare (800), poi con un carattere non numerico (il trattino), poi con un gruppo di tre cifre da memorizzare (555), poi con un carattere non numerico (il trattino), poi con un gruppo di quattro cifre da memorizzare (1212), poi con zero caratteri non numerici, poi con un gruppo di zero cifre da memorizzare, poi con la fine della stringa.
4. Questa è una delle situazioni in cui le espressioni regolari mi fanno venire voglia di cavarmi gli occhi con un oggetto spuntato. Perché questo numero di telefono non corrisponde? Perché c'è un 1 prima del codice d'area, ma avete supposto che tutti i caratteri prima del codice d'area fossero caratteri non numerici (`\D*`). Aargh.

Rivediamo il nostro lavoro per un secondo. Finora le espressioni regolari hanno tutte trovato una corrispondenza dall'inizio della stringa. Ma ora scoprite che all'inizio della stringa potrebbe esserci una quantità indeterminata di caratteri che volete ignorare. Piuttosto che provare a trovare una corrispondenza con tutta quella roba solo per scartarla, adottiamo un approccio differente: evitiamo di cercare esplicitamente una corrispondenza con l'inizio della stringa. Questo approccio è mostrato nel prossimo esempio.

```

>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ③
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234').groups() ④
('800', '555', '1212', '1234')

```

1. Notate la mancanza di ^ in questa espressione regolare. Non state più trovando una corrispondenza con l'inizio della stringa. Non c'è niente che dica che avete bisogno di trovare una corrispondenza tra l'intera stringa in ingresso e la vostra espressione regolare. Il riconoscitore di espressioni regolari si sobbarcherà il duro lavoro necessario a capire quando la stringa in ingresso comincia a corrispondere e proseguirà da lì.
2. Ora potete riconoscere con successo un numero telefonico che include caratteri iniziali e una cifra iniziale, più un qualsiasi numero di qualsiasi tipo di separatori attorno a ogni parte del numero telefonico.
3. La prova del nove. Funziona ancora.
4. Anche questo funziona ancora.

Avete visto quanto velocemente un'espressione regolare può finire fuori controllo? Date una rapida occhiata a una qualsiasi delle iterazioni precedenti. Siete in grado di trovare la differenza tra quella e la successiva?

Mentre avete ancora ben chiara la risposta finale (ed è la risposta finale; se avete scoperto un caso che non gestisce, non voglio saperlo) scriviamola come un'espressione regolare verbosa, prima di dimenticare perché avete fatto le scelte che avete fatto.

```
>>> phonePattern = re.compile(r"""
    # non partire dall'inizio della stringa, il numero può essere ovunque
    (\d{3})    # il codice d'area è di 3 cifre (e.g. '800')
    \D*       # separatore opzionale composto da caratteri che non sono cifre
    (\d{3})    # il prefisso è di 3 cifre (e.g. '555')
    \D*       # separatore opzionale
    (\d{4})    # il resto del numero è di 4 cifre (e.g. '1212')
    \D*       # separatore opzionale
    (\d*)      # l'estensione opzionale è di un qualsiasi numero di cifre
    $         # fine della stringa
    """, re.VERBOSE)

>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups() ①
('800', '555', '1212', '1234')

>>> phonePattern.search('800-555-1212') ②
('800', '555', '1212', '')
```

1. A parte il fatto di essere distribuita su più righe, questa è esattamente l'espressione regolare dell'ultimo passo, quindi non ci sorprende che riconosca gli stessi ingressi.
2. La prova del nove conclusiva. Sì, funziona ancora. Avete finito.



5.7. RIEPILOGO

Questa è solo la più piccola punta dell'iceberg di quello che le espressioni regolari possono fare. In altre parole, anche se ora ne siete completamente sommersi, credetemi, non avete ancora visto niente.

Ora dovrete avere familiarità con le seguenti tecniche:

- `^` corrisponde all'inizio di una stringa;
- `$` corrisponde alla fine di una stringa;
- `\b` corrisponde al limite di una parola;

- `\d` corrisponde a qualsiasi cifra numerica;
- `\D` corrisponde a qualsiasi carattere non numerico;
- `x?` corrispondere a un carattere `x` opzionale (in altre parole, corrisponde a una `x` zero o una volta);
- `x*` corrisponde a `x` zero o più volte;
- `x+` corrisponde a `x` una o più volte;
- `x{n,m}` corrisponde a un carattere `x` almeno `n` volte, ma non più di `m` volte;
- `(a|b|c)` corrisponde esattamente a uno tra `a`, `b`, o `c`;
- `(x)` in generale è un *gruppo di memorizzazione*. Potete ottenere il valore di ciò che vi corrisponde usando il metodo `groups()` dell'oggetto restituito da `re.search()`.

Le espressioni regolari sono estremamente potenti, ma non sono la soluzione corretta per tutti i problemi. Dovreste imparare a conoscerle abbastanza da sapere quando sono appropriate, quando risolveranno i vostri problemi, e quando causeranno più problemi di quelli che risolvono.

CAPITOLO 6. CHIUSURE & GENERATORI

“ *La mia ortografia è Incerta. Non è una cattiva ortografia, ma non è Certa, e le lettere vanno nei posti sbagliati.* ”

— Winnie-the-Pooh

6.1. IMMERSIONE!

Ho sempre subito il fascino dei linguaggi, da degno figlio di un bibliotecario e di una laureata in letteratura inglese. Non parlo di linguaggi di programmazione. Be' sì, parlo di linguaggi di programmazione, ma anche di linguaggi naturali. Prendete l'inglese. L'inglese è una lingua schizofrenica che prende in prestito parole dal tedesco, francese, spagnolo e latino (per nominarne alcune). In realtà, “prende a prestito” è l'espressione sbagliata; “saccheggia” è più corretto. O forse “assimila” — come i Borg. Sì, mi piace.

Noi siamo i Borg. Assimileremo le vostre peculiarità linguistiche ed etimologiche alle nostre. La resistenza è inutile.

In questo capitolo, imparerete qualcosa sui sostantivi plurali in inglese. E anche sulle funzioni che restituiscono altre funzioni, sull'uso avanzato delle espressioni regolari e sui generatori. Ma prima, parliamo di come si costruiscono i sostantivi plurali in inglese. (Se non avete ancora letto il capitolo sulle espressioni regolari, questo potrebbe essere un buon momento per farlo. Questo capitolo presume che abbiate capito le basi delle espressioni regolari, e si addentrerà molto presto nelle loro tecniche più avanzate.)

Se siete cresciuti in un paese di lingua madre inglese o avete imparato l'inglese in un ambiente scolastico formale, avrete familiarità con le regole di base.

- Se una parola finisce con S, X, o Z, aggiungete ES. *Bass* diventa *basses*, *fax* diventa *faxes* e *waltz* diventa *waltzes*.
- Se una parola finisce con una H sonora, aggiungete ES; se finisce con una H muta, aggiungete solo S. Che cos'è una H sonora? È una H che viene combinata con altre lettere per creare un suono che si possa udire.

Quindi *coach* diventa *coaches* e *rash* diventa *rashes*, perché potete udire i suoni /tʃ/ e /ʃ/ quando li pronunciate.

Ma *cheetah* diventa *cheetahs*, perché la H è muta.

- Se una parola finisce con una Y che ha un suono /i/, cambiate la Y in IES; se la Y è combinata con una vocale per ottenere un altro suono, aggiungete solo S. Così *vacancy* diventa *vacancies*, ma *day* diventa *days*.
- Se tutto il resto fallisce, aggiungete S e sperate per il meglio.

(Lo so, ci sono un sacco di eccezioni. *Man* diventa *men* e *woman* diventa *women*, ma *human* diventa *humans*. *Mouse* diventa *mice* e *louse* diventa *lice*, ma *house* diventa *houses*. *Knife* diventa *knives* e *wife* diventa *wives*, ma *lowlife* diventa *lowlifes*. E non fatemi parlare delle invariabili, parole che sono il loro stesso plurale come *sheep*, *deer* e *haiku*.)

In altre lingue, ovviamente, le cose sono completamente differenti.

Progettiamo una libreria Python per pluralizzare automaticamente i sostantivi inglesi. Cominceremo giusto con queste quattro regole, ma tenete a mente che dovrete inevitabilmente aggiungerne altre.



6.2. HO CAPITO, USIAMO LE ESPRESSIONI REGOLARI!

E così state esaminando le parole, il che significa, almeno in inglese, che state esaminando stringhe di caratteri. Avete regole che dicono che dovete trovare combinazioni differenti di caratteri e poi operare su quelle combinazioni in modi differenti. Questo sembra un lavoro per le espressioni regolari!


```

import re

def plural(noun):
    if re.search('[sxz]$', noun):           ①
        return re.sub('$', 'es', noun)      ②
    elif re.search('[^aeiou]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'

```

1. Questa è un'espressione regolare, ma usa una sintassi che non avete visto nel capitolo *Espressioni regolari*. Le parentesi quadre significano “trova una corrispondenza con esattamente uno di questi caratteri”. Così [sxz] significa “s, oppure x, oppure z”, ma solo uno di loro. Il simbolo \$ dovrebbe risultarvi familiare: corrisponde alla fine della stringa. Combinata, questa espressione regolare verifica che noun finisca con s, x, oppure z.
2. Questa funzione re.sub() esegue sostituzioni di stringhe sulla base di un'espressione regolare.

Diamo un'occhiata più dettagliata alle sostituzioni di espressioni regolari.

```

>>> import re

>>> re.search('[abc]', 'Mark')           ①
<_sre.SRE_Match object at 0x001C1FA8>

>>> re.sub('[abc]', 'o', 'Mark')         ②
'Mork'

>>> re.sub('[abc]', 'o', 'rock')          ③
'rook'

>>> re.sub('[abc]', 'o', 'caps')          ④
'oops'

```

1. La stringa Mark contiene a, b, oppure c? Sì, contiene a.
2. OK, adesso trova a, b, oppure c e sostituiscilo con o. Mark diventa Mork.
3. La stessa funzione trasforma rock in rook.

4. Potreste pensare che questo trasformi caps in oaps, ma non lo fa. `re.sub()` sostituisce *tutte* le corrispondenze, non solo la prima. Quindi questa espressione regolare trasforma caps in oops, perché sia la *c* che la *a* vengono trasformate in *o*.

E ora, torniamo alla funzione `plural()`...

```
def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)          ①
    elif re.search('[^aeiou]h$', noun):         ②
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):         ③
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

1. Qui state sostituendo la fine della stringa (a cui corrisponde il simbolo `$`) con la stringa `es`. In altre parole, state aggiungendo `es` alla stringa. Potreste ottenere la stessa cosa con la concatenazione di stringhe, per esempio `noun + 'es'`, ma ho scelto di usare le espressioni regolari in tutte le regole per ragioni che diverranno chiare più avanti in questo capitolo.
2. Guardate attentamente, questa è un'altra nuova variazione. Il simbolo `^` come primo carattere tra parentesi quadre significa qualcosa di speciale: negazione. `[^abc]` significa “ogni singolo carattere *tranne* a, b, oppure c”. Allo stesso modo, `[^aeiou]h$` significa qualsiasi carattere *tranne* a, e, i, o, u, d, g, k, p, r, oppure t. Successivamente, quel carattere deve essere seguito da *h*, seguito a sua volta dalla fine della stringa. State cercando parole che finiscono in *H* dove la *H* può essere udita.
3. Stesso schema qui: l'espressione regolare corrisponde a parole che finiscono in *Y*, dove il carattere che precede la *Y* *non* è a, e, i, o, oppure u. State cercando parole che finiscono con una *Y* che ha un suono /i/.

Diamo una occhiata più dettagliata alla negazione nelle espressioni regolari.

```
>>> import re

>>> re.search('[^aeiou]y$', 'vacancy') ①
<_sre.SRE_Match object at 0x001C1FA8>

>>> re.search('[^aeiou]y$', 'boy')      ②
>>>
>>> re.search('[^aeiou]y$', 'day')
>>>

>>> re.search('[^aeiou]y$', 'pita')     ③
>>>
```

1. vacancy corrisponde a questa espressione regolare perché finisce con cy e c non è a, e, i, o, oppure u.
2. boy non corrisponde perché finisce con oy e avete specificato che il carattere prima della y non può essere o. day non corrisponde perché termina in ay.
3. pita non corrisponde perché non termina in y.

```
>>> re.sub('y$', 'ies', 'vacancy')      ①
'vacancies'

>>> re.sub('y$', 'ies', 'agency')
'agencies'

>>> re.sub('([^\aeiou])y$', r'\1ies', 'vacancy') ②
'vacancies'
```

1. Questa espressione regolare trasforma vacancy in vacancies e agency in agencies, che è quello che volevate. Notate che trasformerebbe anche boy in boies, ma questo non succederà mai nella funzione plural() perché avete utilizzato re.search() per capire se è il caso di applicare re.sub() o meno.
2. Giusto di passaggio, voglio sottolineare che è possibile combinare queste due espressioni regolari (una per scoprire se la regola è applicabile, l'altra per applicarla effettivamente) in una singola espressione regolare. Questa riga mostra come apparirebbe l'espressione combinata. La maggior parte dell'espressione dovrebbe sembrarvi familiare: state usando un gruppo di memorizzazione, che avete imparato nella sezione Caso di studio: riconoscere i numeri di telefono. Qui il gruppo viene usato per memorizzare il carattere prima della lettera y. Poi, nella stringa di sostituzione usate una nuova sintassi, \1, che significa "ehi, hai presente quel primo gruppo che hai memorizzato? Mettilo proprio qui." In questo caso, avete memorizzato la c prima della y, perciò quando effettuate la sostituzione mettete c al posto di c e ies al posto di y. (Se avete memorizzato più di un gruppo, potete usare \2 e \3 e così via.)

Le sostituzioni di espressioni regolari sono estremamente potenti, e la sintassi \1 le rende ancora più potenti. Ma combinare entrambe le operazioni in un'unica espressione regolare è anche più difficile da leggere, e non corrisponde direttamente al modo in cui avete precedentemente descritto le regole per la pluralizzazione. Avete originariamente esposto le regole come “se la parola finisce con S, X, o Z, allora aggiungete ES”. Se guardate la funzione `plural()`, ci sono due righe di codice che dicono esattamente “se la parola finisce con S, X, o Z, allora aggiungete ES”. Non potete essere più diretti di così.



6.3. UNA LISTA DI FUNZIONI

Ora state per aggiungere un livello di astrazione. Avete cominciato definendo una lista di regole: se accade questo fai quello, altrimenti vai alla prossima regola. Complichiamo temporaneamente parte del programma in modo da semplificarne un'altra parte.

```

import re

def match_sxz(noun):
    return re.search('[sxz]$', noun)

def apply_sxz(noun):
    return re.sub('$', 'es', noun)

def match_h(noun):
    return re.search('[^aeiou]h$', noun)

def apply_h(noun):
    return re.sub('$', 'es', noun)

def match_y(noun):
    return re.search('[^aeiou]y$', noun)

def apply_y(noun):
    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return True

def apply_default(noun):
    return noun + 's'

rules = ((match_sxz, apply_sxz),
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
        )

def plural(noun):
    for matches_rule, apply_rule in rules:

```

```
if matches_rule(noun):  
    return apply_rule(noun)
```

1. Ora, ogni regola di ricerca è contenuta in una propria funzione che restituisce i risultati della chiamata alla funzione `re.search()`.
2. Anche ogni regola di sostituzione è contenuta in una propria funzione che chiama la funzione `re.sub()` per applicare la regola di pluralizzazione appropriata.
3. Invece di avere una funzione (`plural()`) che contiene più regole, avete la struttura dati `rules` che è una sequenza di coppie di funzioni.
4. Dato che le regole sono state estratte in una struttura dati separata, la nuova funzione `plural()` può essere ridotta a poche righe di codice. Usando un ciclo `for`, potete estrarre regole di ricerca e sostituzione due alla volta (una per il primo tipo, una per il secondo) dalla struttura dati `rules`. Alla prima iterazione del ciclo `for`, `matches_rule` varrà `match_sxz` e `apply_rule` varrà `apply_sxz`. Alla seconda iterazione (supponendo che ci arrivate), a `matches_rule` verrà assegnata `match_h` e ad `apply_rule` verrà assegnata `apply_h`. La funzione garantisce di restituire qualcosa alla fine, perché l'ultima regola di ricerca (`match_default`) restituisce semplicemente `True`, indicando che la regola di sostituzione corrispondente (`apply_default`) sarà sempre applicata.

La ragione per cui questa tecnica funziona è che ogni cosa in Python è un oggetto, comprese le funzioni. La struttura dati `rules` contiene funzioni — non nomi di funzioni, ma veri e propri oggetti funzione. Quando questi oggetti vengono assegnati nel ciclo `for`, allora `matches_rule` e `apply_rule` diventano vere e proprie funzioni che potete eseguire. La prima iterazione del ciclo `for` è equivalente a invocare `matches_sxz(noun)` e, se viene restituita una corrispondenza, a invocare anche `apply_sxz(noun)`.

Se questo livello di astrazione aggiuntivo vi confonde, provate a sviluppare la funzione per vedere l'equivalenza. L'intero ciclo `for` è equivalente a quanto segue:

*La variabile
“rules” è una
sequenza di*

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

*coppie di
funzioni.*

Il beneficio qui è che questa funzione `plural()` è ora semplificata. Prende una sequenza di regole definite da qualche altra parte e itera attraverso di loro in modo generico.

1. Ottiene una regola di ricerca.
2. Trova una corrispondenza? Allora chiama la regola di sostituzione e restituisce il risultato.
3. Nessuna corrispondenza? Ritorna al passo 1.

Le regole potrebbero essere definite ovunque, in ogni caso. La funzione `plural()` non se ne cura.

Ora, valeva la pena aggiungere questo livello di astrazione? Be', non ancora. Consideriamo cosa bisognerebbe fare per aggiungere una nuova regola alla funzione. Nel primo esempio, questo richiederebbe l'aggiunta di un'istruzione `if` alla funzione `plural()`. Nel secondo esempio, richiederebbe l'aggiunta di due funzioni, `match_foo()` e `apply_foo()`, e l'aggiornamento della sequenza `rules` per specificare in quale ordine le nuove funzioni di ricerca e sostituzione dovrebbero essere chiamate rispetto alle altre regole.

Ma questa è solo una pietra miliare verso la prossima sezione. Proseguiamo...

*
**

6.4. UNA LISTA DI PATTERN

Definire separatamente una funzione con un proprio nome per ogni regola di ricerca e sostituzione non è realmente necessario. Non le chiamate mai direttamente, ma le aggiungete alla sequenza `rules` e le chiamate attraverso di essa. In più, ogni funzione segue uno di due schemi. Tutte le funzioni di ricerca chiamano `re.search()` e tutte le funzioni di sostituzione chiamano `re.sub()`. Evidenziamo gli schemi in modo che definire nuove regole diventi più semplice.

```
import re

def build_match_and_apply_functions(pattern, search, replace):
    def matches_rule(word): ①
        return re.search(pattern, word)
    def apply_rule(word): ②
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule) ③
```

1. `build_match_and_apply_functions()` è una funzione che costruisce dinamicamente altre funzioni. Accetta in ingresso `pattern`, `search` e `replace`, poi definisce una funzione `matches_rule()` che chiama `re.search()` con il `pattern` che è stato passato alla funzione `build_match_and_apply_functions()` e con la parola `word` che è stata passata alla funzione `matches_rule()` che state costruendo. Whoa.
2. La funzione di sostituzione viene costruita allo stesso modo. Accetta un parametro e chiama `re.sub()` utilizzando sia i parametri `search` e `replace` che sono stati passati alla funzione `build_match_and_apply_functions()` sia la parola `word` che è stata passata alla funzione `apply_rule()` stessa. Questa tecnica di usare i valori di parametri esterni all'interno di una funzione dinamica è chiamata *chiusura*. State essenzialmente definendo delle costanti nell'ambito della funzione di sostituzione che state costruendo; quest'ultima accetta un parametro (`word`), ma poi agisce su quello più altri due valori (`search` e `replace`) che sono stati impostati nel momento in cui la funzione di sostituzione è stata definita.
3. Infine, la funzione `build_match_and_apply_functions()` restituisce una tupla di due valori: le due funzioni che avete appena creato. Le costanti definite in quelle funzioni (`pattern` nell'ambito di `matches_rule()`, `search` e `replace` nell'ambito di `apply_rule()`) mantengono il loro valore in quelle funzioni persino dopo che `build_match_and_apply_functions()` le ha restituite. Questo è davvero fantastico.

Se questa tecnica vi risulta incredibilmente confusa (e dovrebbe, è roba strana), potrebbe diventare più chiara nel momento in cui vedete come si usa.

```
patterns = \                                ①
(
    ('[sxz]$',          '$', 'es'),
    ('^[aeioudgkprt]h$', '$', 'es'),
    ('(qu|^[aeiou])y$', 'y$', 'ies'),
    ('$',               '$', 's')           ②
)
rules = [build_match_and_apply_functions(pattern, search, replace) ③
         for (pattern, search, replace) in patterns]
```

1. Le nostre “regole” di pluralizzazione sono ora definite come una tupla di tuple di *stringhe* (non di funzioni). La prima stringa in ogni gruppo è l’espressione regolare che usereste in `re.search()` per vedere se c’è una corrispondenza con questa regola. La seconda e la terza stringa di ogni gruppo sono le espressioni di ricerca e sostituzione che usereste in `re.sub()` per applicare effettivamente la regola e trasformare un sostantivo nel suo plurale.
2. C’è un leggero cambiamento qui, nella regola di ripiego. Nell’esempio precedente, la funzione `match_default()` restituiva semplicemente `True`, indicando che, se nessuna delle regole più specifiche trovava una corrispondenza, il codice avrebbe semplicemente aggiunto una *s* alla fine della parola *data*. Questo esempio fa qualcosa di funzionalmente equivalente. L’ultima espressione regolare chiede se la parola ha una fine (`$` corrisponde alla fine di una stringa). Naturalmente, ogni stringa ha una fine, anche la stringa vuota, quindi questa espressione trova sempre una corrispondenza. Quindi, questa espressione regolare serve allo stesso scopo della funzione `match_default()` che restituiva sempre `True`: si assicura che, se nessuna regola più specifica trova una corrispondenza, il codice aggiunga una *s* alla fine della parola *data*.
3. Questa riga è magia. Prende la sequenza di stringhe in `patterns` e la trasforma in una sequenza di funzioni. Come? “Correlando” le stringhe con il valore di ritorno della funzione `build_match_and_apply_functions()`. Cioè, prende ogni tripla di stringhe e invoca la funzione `build_match_and_apply_functions()` con quelle tre stringhe come argomenti. La funzione `build_match_and_apply_functions()` restituisce una tupla di due funzioni. Questo significa che `rules` finisce per essere funzionalmente equivalente a ciò che era nell’esempio precedente: una lista di tuple, dove ogni tupla contiene una coppia di funzioni. La prima funzione è la funzione di ricerca che chiama `re.search()` e la seconda funzione è la funzione di sostituzione che chiama `re.sub()`.

Per completare questa versione del programma ecco il punto di entrata principale, la funzione `plural()`.

```
def plural(noun):  
    for matches_rule, apply_rule in rules: ①  
        if matches_rule(noun):  
            return apply_rule(noun)
```

- I. Visto che la lista `rules` è la stessa dell'esempio precedente (davvero, lo è), non dovrebbe sorprendervi che la funzione `plural()` non sia cambiata per niente. Rimane completamente generica; prende una lista di funzioni di regole e le chiama in ordine, senza preoccuparsi di come le regole sono definite. Nell'esempio precedente, le regole erano definite come funzioni separate con un proprio nome. Ora sono costruite dinamicamente attraverso la correlazione tra una lista di stringhe e il valore di ritorno della funzione `build_match_and_apply_functions()`. Non ha importanza; la funzione `plural()` continua a lavorare nello stesso modo.



6.5. UN FILE DI PATTERN

Avete rimosso tutto il codice duplicato e avete aggiunto abbastanza astrazioni per definire le regole di pluralizzazione in una lista di stringhe. Il passo logico successivo consiste nel prendere queste stringhe e metterle in un file separato, dove possano essere mantenute separatamente dal codice che le usa.

Prima di tutto, creiamo un file di testo che contiene le regole che volete. Nessuna struttura dati elaborata, solo stringhe separate da spazi bianchi su tre colonne. Chiamiamo questo file `plural4-rules.txt`.

```
[sxz]$          $    es  
[^aeioudgkprt]h$ $    es  
[^aeiou]y$      y$   ies  
$               $    s
```

Ora vediamo come potete usare questo file di regole.

```

import re

def build_match_and_apply_functions(pattern, search, replace): ①
    def matches_rule(word):
        return re.search(pattern, word)
    def apply_rule(word):
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule)

rules = []

with open('plural4-rules.txt', encoding='utf-8') as pattern_file: ②
    for line in pattern_file: ③
        pattern, search, replace = line.split(None, 3) ④
        rules.append(build_match_and_apply_functions( ⑤
            pattern, search, replace))

```

1. La funzione `build_match_and_apply_functions()` non è cambiata. State ancora sfruttando le chiusure per costruire dinamicamente due funzioni che usano variabili definite nella funzione esterna.
2. La funzione globale `open()` apre un file e restituisce un oggetto file. In questo caso, il file che stiamo aprendo contiene le stringhe dei pattern per la pluralizzazione dei sostantivi. L'istruzione `with` crea quello che viene chiamato un *contesto*: quando il blocco `with` finisce, Python chiuderà automaticamente il file, anche se è stata sollevata un'eccezione all'interno del blocco `with`. Imparerete di più sui blocchi `with` e sugli oggetti file nel capitolo [File](#).
3. L'idioma `for line in <oggettofile>` legge dati da un file aperto, una riga alla volta, e assegna il testo alla variabile `line`. Imparerete di più su come leggere i file nel capitolo [File](#).
4. Ogni riga nel file in realtà contiene tre valori, separati da spazi bianchi (che siano tabulazioni o spazi non fa alcuna differenza). Per recuperarli, usate il metodo `split()` delle stringhe. Il primo argomento del metodo `split()` è `None`, che significa “spezza la stringa in corrispondenza di qualsiasi spazio bianco (che siano tabulazioni o spazi non fa alcuna differenza)”. Il secondo argomento è `3`, che significa “spezza la stringa in corrispondenza di spazi bianchi per 3 volte, poi lascia il resto così com'è”. Una riga come `[sxz]$ $ es` verrà suddivisa nella lista `['[sxz]$', '$', 'es']`, il che significa che `pattern` avrà il valore `'[sxz]$',` `search` avrà il valore `'$'` e `replace` avrà il valore `'es'`. Questa piccola riga di codice è molto potente.

5. Infine, passate `pattern`, `search` e `replace` alla funzione `build_match_and_apply_functions()`, che restituisce una tupla di funzioni. Aggiungete questa tupla in coda alla lista `rules` e `rules` finirà per memorizzare la lista delle funzioni di ricerca e sostituzione che la funzione `plural()` si aspetta.

In questo caso il miglioramento è che avete completamente isolato le regole di pluralizzazione in un file esterno, in modo che possa essere gestito separatamente dal codice che lo utilizza. Il codice è codice, i dati sono dati, e la vita è bella.

**
**

6.6. GENERATORI

Non sarebbe magnifico avere una funzione `plural()` generica che analizza il file di regole? Ottieni le regole, controlla se c'è una corrispondenza, applica la trasformazione appropriata, vai alla regola successiva. Questo è tutto quello che la funzione `plural()` ha bisogno di fare, e questo è tutto quello che la funzione `plural()` dovrebbe fare.

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3)
            yield build_match_and_apply_functions(pattern, search, replace)

def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename):
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {}'.format(noun))
```

Come cavolo funziona *questo*? Diamo prima un'occhiata a un esempio nella shell interattiva.

```

>>> def make_counter(x):
...     print('entro in make_counter')
...     while True:
...         yield x                ①
...         print('incremento x')
...         x = x + 1
...
>>> counter = make_counter(2)      ②
>>> counter                        ③
<generator object at 0x001C9C10>
>>> next(counter)                  ④
entro in make_counter
2
>>> next(counter)                  ⑤
incremento x
3
>>> next(counter)                  ⑥
incremento x
4

```

1. La presenza della parola chiave `yield` in `make_counter()` significa che questa non è una normale funzione. È un tipo speciale di funzione che genera valori uno alla volta. Potete pensarla come una funzione riavviabile. L'atto di chiamarla vi restituirà un *generatore* che può essere usato per generare valori successivi di `x`.
2. Per creare un'istanza del generatore `make_counter()`, vi basta invocarlo come ogni altra funzione. Notate che la chiamata non esegue effettivamente il codice della funzione. Potete rendervene conto perché la prima riga della funzione `make_counter()` invoca `print()`, ma nulla è stato ancora stampato.
3. La funzione `make_counter()` restituisce un oggetto generatore.
4. La funzione `next()` accetta un generatore e restituisce il suo valore successivo. La prima volta che chiamate `next()` con il generatore `counter`, il codice in `make_counter()` viene eseguito fino alla prima istruzione `yield`, poi restituisce il valore che è stato generato. In questo caso, quel valore sarà 2, perché avete originariamente creato il generatore invocando `make_counter(2)`.
5. Invocare ripetutamente `next()` con lo stesso oggetto generatore fa riprendere l'esecuzione esattamente da dove era rimasta sospesa, proseguendo fino ad arrivare all'istruzione `yield` successiva. Tutte le variabili, lo stato locale, &c. sono salvate da `yield` e ripristinate da `next()`. La successiva riga di codice che attende di

essere eseguita chiama `print()`, che stampa incremento `x`. Dopodiché, viene eseguita l'istruzione `x = x + 1`. Poi si passa di nuovo attraverso il ciclo `while` e la prima cosa che si esegue è l'istruzione `yield x`, che salva lo stato di tutto e restituisce il valore corrente di `x` (ora 3).

6. La seconda volta che chiamate `next(counter)` fate di nuovo tutte le stesse cose, ma questa volta `x` vale 4

Dato che `make_counter()` contiene un ciclo infinito, potreste teoricamente andare avanti in questo modo per sempre, e il generatore continuerebbe semplicemente a incrementare `x` e a produrre valori in uscita. Ma diamo invece un'occhiata a usi più produttivi dei generatori.

6.6.1. UN GENERATORE DI FIBONACCI

```
def fib(max):  
    a, b = 0, 1          ①  
    while a < max:  
        yield a          ②  
        a, b = b, a + b   ③
```

1. La sequenza di Fibonacci è una sequenza numerica in cui ogni numero è la somma dei due numeri che lo precedono. Comincia con 0 e 1, si incrementa lentamente all'inizio, poi sempre più rapidamente. Per cominciare la sequenza avete bisogno di due variabili: `a` parte da 0 e `b` parte da 1.
2. `a` è il numero corrente nella sequenza, quindi producetelo in uscita.
3. `b` è il numero successivo nella sequenza, quindi assegnatelo ad `a`, ma calcolate anche il prossimo valore (`a + b`) e assegnatelo a `b` per usarlo più tardi. Notate che questo accade in parallelo: se `a` è 3 e `b` è 5, allora `a, b = b, a + b` imposterà `a` a 5 (il valore precedente di `b`) e `b` a 8 (la somma dei valori precedenti di `a` e `b`).

*“yield”
sospende una
funzione.
“next()” la
riavvia da
dove era
rimasta
sospesa.*

Quindi avete una funzione che produce in uscita numeri di Fibonacci successivi. Certo, potreste farlo con la ricorsione, ma in questo modo è più facile da leggere. In più, funziona bene con i cicli `for`.

```
>>> from fibonacci import fib
>>> for n in fib(1000):      ①
...     print(n, end=' ')    ②
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> list(fib(1000))         ③
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

1. Potete usare un generatore come `fib()` direttamente in un ciclo `for`. Il ciclo `for` chiamerà automaticamente la funzione `next()` per ottenere valori dal generatore `fib()` e assegnarli alla variabile di indice del ciclo `for` (cioè `n`).
2. A ogni iterazione nel ciclo `for`, `n` ottiene un nuovo valore dall'istruzione `yield` in `fib()`, e tutto quello che dovete fare è stamparlo. Una volta che `fib()` ha finito i numeri (a diventa più grande di `max`, che nel nostro caso è 1000) allora il ciclo `for` si conclude normalmente.
3. Questo è un utile idiomma: passate un generatore alla funzione `list()` e la funzione itererà attraverso l'intero generatore (esattamente come il ciclo `for` nell'esempio precedente) per poi restituire una lista di tutti i valori.

6.6.2. UN GENERATORE DI REGOLE PER I SOSTANTIVI PLURALI

Torniamo indietro a `plural5.py` e vediamo se questa versione della funzione `plural()` funziona.

```

def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3) ①
            yield build_match_and_apply_functions(pattern, search, replace) ②

def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename): ③
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('nessuna regola per {}'.format(noun))

```

1. Nessuna magia qui. Ricordate che le righe del file di regole contengono tre valori separati da spazi bianchi, quindi usate `line.split(None, 3)` per ottenere le tre “colonne” e assegnarle a tre variabili locali.
2. *E poi utilizzate yield.* Che cosa produce? Due funzioni, costruite dinamicamente con la vostra vecchia amica `build_match_and_apply_functions()` che è identica all’esempio precedente. In altre parole, `rules()` è un generatore che produce funzioni di ricerca e sostituzione *su richiesta*.
3. Dato che `rules()` è un generatore, potete usarlo direttamente in un ciclo `for`. Alla prima iterazione nel ciclo `for` chiamerete la funzione `rules()`, che aprirà il file dei pattern, leggerà la prima riga, costruirà dinamicamente una funzione di ricerca e una funzione di sostituzione usando i pattern di quella riga e produrrà in uscita le funzioni costruite dinamicamente. Alla seconda iterazione nel ciclo `for`, `rules()` riprenderà l’esecuzione esattamente da dove era rimasta (cioè nel mezzo del ciclo `for line in pattern_file`). Come prima cosa leggerà la riga successiva del file (che è ancora aperto), poi costruirà dinamicamente un’altra coppia di funzioni di ricerca e sostituzione basate sui pattern di quella riga, infine produrrà in uscita le due funzioni.

Che cosa avete guadagnato rispetto alla versione 4? Tempo di inizializzazione. Nella versione 4, quando importavate il modulo `plural4`, il file dei pattern veniva interamente letto e una lista di tutte le possibili regole veniva costruita prima ancora che poteste anche solo pensare di chiamare la funzione `plural()`. Con i generatori, potete fare ogni cosa in maniera ritardata: leggete la prima regola, create le corrispettive funzioni e le provate, e se la prima regola è quella giusta non dovete nemmeno leggere il resto del file o creare altre funzioni.

Che cosa avete perso? Prestazioni! Ogni volta che chiamate la funzione `plural()`, il generatore `rules()` ricomincia dall'inizio — il che vuol dire riaprire il file dei pattern e leggerlo dall'inizio, una riga alla volta.

E se poteste avere il meglio dei due mondi? Minimo costo di inizializzazione (evitando di eseguire codice alla chiamata di `import`) e massime prestazioni (evitando di costruire sempre le stesse funzioni ogni volta). Oh, e volete ancora mantenere le regole in un file separato (perché il codice è codice e i dati sono dati), fino a quando non dobbiate leggere la stessa riga due volte.

Per farlo, avrete bisogno di costruire un vostro iteratore. Ma prima che possiate fare *questo*, dovete imparare le classi Python.



6.7. LETTURE DI APPROFONDIMENTO

- PEP 255: Generatori semplici
- Capire l'istruzione "with" di Python
- Le chiusure in Python
- I numeri di Fibonacci
- I sostantivi plurali irregolari in inglese

CAPITOLO 7. CLASSI & ITERATORI

“ L’Oriente è l’Oriente e l’Occidente è l’Occidente, e mai i due si incontreranno. ”

— Rudyard Kipling

7.1. IMMERSIONE!

In verità, gli iteratori sono l’“ingrediente segreto” di Python 3: si trovano ovunque, nascosti dietro le quinte di ogni funzionalità, sempre appena fuori vista. Le descrizioni sono solo un caso speciale di *iteratori*. Anche i generatori sono solo un caso speciale di *iteratori*: una funzione che produce valori tramite `yield` è un modo gradevole e compatto di costruire un iteratore senza costruire un iteratore. Lasciate che vi mostri cosa voglio dire.

Ricordate il generatore di Fibonacci? Qui di seguito lo trovate reimplementato da zero sotto forma di iteratore:

```

class Fib:
    '''iteratore che produce i numeri della sequenza di Fibonacci'''
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib

```

Analizziamolo una riga alla volta.

```

class Fib:

```

class? Classe? Che cos'è una classe?

*
 **

7.2. DEFINIRE LE CLASSI

Python è completamente orientato agli oggetti: potete definire le vostre classi, ereditare dalle vostre classi o dalle classi built-in e istanziare le classi che avete definito.


Definire una classe in Python è semplice. Come per le funzioni, non c'è una definizione di interfaccia separata. Vi basta definire la classe e cominciare a programmarla. Una classe Python comincia con la parola

riservata `class`, seguita dal nome della classe. Tecnicamente, questo è tutto quello che è richiesto, in quanto una classe non ha bisogno di ereditare da nessun'altra classe.

```
class PapayaWhip: ①  
    pass          ②
```

1. Il nome di questa classe è `PapayaWhip`, e non eredita da nessun'altra classe. I nomi delle classi solitamente cominciano con una lettera maiuscola, `OgniParolaInQuestoModo`, ma questa è solo una convenzione, non un obbligo.
2. Lo avrete probabilmente già indovinato, ma ogni cosa in una classe è indentata, esattamente come il codice all'interno di una funzione, di un'istruzione `if`, di un ciclo `for`, o di ogni altro blocco di codice. La prima riga non indentata non fa più parte della definizione di classe.

Questa classe `PapayaWhip` non definisce alcun metodo o attributo, ma sintatticamente la definizione deve contenere qualcosa, ed ecco il motivo per la presenza di quella istruzione `pass`. Questa è una parola riservata di Python che significa solo “circolate, non c'è nulla da vedere”. È un'istruzione che non fa nulla, e un buon segnaposto da sfruttare quando state definendo lo scheletro di funzioni o classi.

 L'istruzione `pass` in Python è come un insieme vuoto di parentesi graffe (`{}`) in Java o C.

Molte classi ereditano da altre classi, ma questa non lo fa. Molte classi definiscono metodi, ma questa non lo fa. Non c'è nulla che una classe Python debba assolutamente avere, a parte un nome. In particolare, i programmatori C++ potrebbero trovare strano che le classi Python non abbiano costruttori e distruttori espliciti. Sebbene non sia obbligatorio, le classi Python *possono* avere qualcosa di simile a un costruttore: il metodo `__init__()`.

7.2.1. IL METODO `__init__()`

Questo esempio mostra l'inizializzazione della classe `Fib` usando il metodo `__init__()`.

```
class Fib:
    '''iteratore che produce i numeri della sequenza di Fibonacci''' ①

    def __init__(self, max): ②
```

1. Anche le classi possono (e dovrebbero) avere una docstring, esattamente come i moduli e le funzioni.
2. Il metodo `__init__()` viene chiamato immediatamente dopo la creazione di un'istanza della classe. Sarebbe allettante — ma tecnicamente scorretto — dire che questo metodo è il “costruttore” della classe. È allettante perché il metodo somiglia a un costruttore C++ (per convenzione, il metodo `__init__()` è il primo metodo definito nella classe), agisce come un costruttore (è il primo frammento di codice eseguito in un'istanza appena creata della classe) e ha persino un nome che suona come quello di un costruttore. Ma è scorretto perché l'oggetto è già stato costruito nel momento in cui il metodo `__init__()` viene invocato, e un riferimento valido alla nuova istanza della classe è già disponibile.

Il primo argomento di ogni metodo in una classe, compreso il metodo `__init__()`, è sempre un riferimento all'istanza corrente della classe. Per convenzione, questo argomento viene chiamato `self`. Questo argomento occupa il ruolo della parola riservata `this` in C++ o Java, ma `self` non è una parola riservata in Python, bensì semplicemente una convenzione nominale. Nondimeno, per favore non chiamatelo in modi diversi da `self` perché questa è una convenzione molto forte.

Nel metodo `__init__()` `self` si riferisce all'oggetto appena creato; in altri metodi della classe si riferisce all'istanza il cui metodo è stato invocato. Sebbene abbiate bisogno di specificare esplicitamente `self` quando definite il metodo, *non* dovete specificarlo quando chiamate il metodo; Python lo aggiungerà per voi automaticamente.

*
**

7.3. ISTANZIARE LE CLASSI

Istanziare le classi in Python è semplice. Per istanziare una classe vi basta invocare la classe come se fosse una funzione, passando gli argomenti richiesti dal metodo `__init__()`. Il valore di ritorno sarà l'oggetto appena creato.

```

>>> import fibonacci2

>>> fib = fibonacci2.Fib(100) ①


>>> fib ②
<fibonacci2.Fib object at 0x00DB8810>

>>> fib.__class__ ③
<class 'fibonacci2.Fib'>

>>> fib.__doc__ ④
'iteratore che produce i numeri della sequenza di Fibonacci'

```

1. State creando un'istanza della classe `Fib` (definita nel modulo `fibonacci2`) e assegnando l'istanza appena creata alla variabile `fib`. State passando un parametro, `100`, che finirà per diventare l'argomento `max` del metodo `__init__()` in `Fib`.
2. `fib` è ora un'istanza della classe `Fib`.
3. Ogni istanza di una classe ha un attributo built-in chiamato `__class__` che è la classe dell'oggetto. I programmatori Java potrebbero avere familiarità con la classe `Class`, che contiene metodi come `getName()` e `getSuperclass()` per ottenere metainformazioni su un oggetto. In Python, questo tipo di metadati si può ottenere attraverso gli attributi, ma l'idea è la stessa.
4. Potete accedere alla docstring dell'istanza esattamente come con una funzione o un modulo. Tutte le istanze di una classe condividono la stessa docstring.

 In Python, vi basta invocare una classe come se fosse una funzione per creare una nuova istanza della classe. Non c'è nessun operatore `new` esplicito come in C++ o in Java.

*
**

7.4. VARIABILI DI ISTANZA

Sulla riga successiva troviamo:

```
class Fib:
    def __init__(self, max):
        self.max = max ①
```

1. Che cos'è `self.max`? È una variabile di istanza. È completamente separata da `max`, che è passato al metodo `__init__()` come un argomento. `self.max` è “globale” per l'istanza. Questo significa che potete accedervi da altri metodi.

```
class Fib:
    def __init__(self, max):
        self.max = max ①
    .
    .
    .
    def __next__(self):
        fib = self.a
        if fib > self.max: ②
```

1. `self.max` è definito nel metodo `__init__()`...
2. ...e riferito nel metodo `__next__()`.

Le variabili di istanza sono specifiche per un'istanza di una classe. Per esempio, se create due istanze di `Fib` con differenti valori massimi, ognuna manterrà in memoria il proprio valore.

```
>>> import fibonacci2
>>> fib1 = fibonacci2.Fib(100)
>>> fib2 = fibonacci2.Fib(200)
>>> fib1.max
100
>>> fib2.max
200
```



7.5. UN ITERATORE DI FIBONACCI

Ora siete pronti per imparare come si costruisce un iteratore. Un iteratore è semplicemente una classe che definisce un metodo `__iter__()`.

```
class Fib: ①
    def __init__(self, max): ②
        self.max = max

    def __iter__(self): ③
        self.a = 0
        self.b = 1
        return self

    def __next__(self): ④
        fib = self.a
        if fib > self.max:
            raise StopIteration ⑤
        self.a, self.b = self.b, self.a + self.b
        return fib ⑥
```

1. Per costruire un iteratore partendo da zero, `Fib` deve essere una classe, non una funzione.
2. “Invocare” `Fib(max)` vuol dire in realtà creare un’istanza di questa classe e chiamare il suo metodo `__init__()` con `max`. Il metodo `__init__()` salva il valore massimo come una variabile di istanza in modo che altri metodi possano usarlo più tardi.
3. Il metodo `__iter__()` viene invocato ogni volta che qualcuno chiama `iter(fib)`. (Come vedrete fra un minuto, un ciclo `for` chiama questo metodo automaticamente, ma potete chiamarlo anche voi manualmente.) Dopo aver eseguito l’inizializzazione che prepara l’iterazione (in questo caso, impostando i valori di `self.a` e `self.b`, i nostri due contatori), il metodo `__iter__()` può restituire qualsiasi oggetto che implementi un metodo `__next__()`. In questo caso (e nella maggior parte dei casi) `__iter__()` restituisce semplicemente `self`, dato che questa classe implementa il proprio metodo `__next__()`.

Tutti e tre i metodi di classe `__init__`, `__iter__` e `__next__` cominciano e finiscono con una coppia di caratteri di sottolineatura (`_`). Come mai? Non c’è nulla di magico in questo, ma di solito indica che questi metodi sono “metodi speciali”. L’unica cosa “speciale” dei metodi speciali è che non vengono invocati direttamente, ma Python li chiama quando usate qualche altra sintassi sulla classe o su un’istanza della classe. Ulteriori informazioni sui metodi speciali.

4. Il metodo `__next__()` viene invocato ogni volta che qualcuno chiama `next()` su un iteratore di un'istanza di una classe. Questo avrà più senso fra un minuto.
5. Quando il metodo `__next__()` solleva un'eccezione di tipo `StopIteration`, questo segnala al chiamante che l'iterazione è esaurita. A differenza della maggior parte delle eccezioni questa non rappresenta un errore, ma una condizione normale che significa solamente che l'iteratore non ha più valori da generare. Se il chiamante è un ciclo `for`, noterà questa eccezione di tipo `StopIteration` e uscirà normalmente dal ciclo. (In altre parole, sopprimerà l'eccezione.) Questo piccolo tocco di magia è in realtà la chiave per usare gli iteratori nei cicli `for`.
6. Per produrre in uscita il valore successivo, il metodo `__next__()` di un iteratore restituisce semplicemente il valore tramite l'istruzione `return`. Qui `yield` non viene usato; quello è zucchero sintattico che si applica solo quando state usando i generatori. Qui invece state creando il vostro iteratore da zero; quindi, usate `return`.

Non siete ancora completamente confusi? Eccellente. Vediamo come invocare questo iteratore:

```
>>> from fibonacci2 import Fib
>>> for n in Fib(1000):
...     print(n, end=' ')
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Ehi, è esattamente lo stesso! Identico byte per byte al modo di invocare il generatore di Fibonacci (a parte una lettera maiuscola). Ma com'è possibile?

C'è un pizzico di magia nel funzionamento dei cicli `for`. Ecco cosa succede.

- Il ciclo `for` invoca `Fib(1000)`, come mostrato. Questo restituisce un'istanza della classe `Fib`. Chiamiamo questa istanza `fib_inst`.
- Segretamente, e piuttosto ingegnosamente, il ciclo `for` chiama `iter(fib_inst)`, che restituisce un oggetto iteratore. Chiamiamo questo oggetto `fib_iter`. In questo caso `fib_iter == fib_inst`, perché il metodo `__iter__()` restituisce `self` anche se il ciclo `for` non lo sa (o non se ne cura).
- Per “attraversare” l'iteratore, il ciclo `for` chiama `next(fib_iter)`, che chiama il metodo `__next__()` sull'oggetto `fib_iter`, che calcola il successivo numero di Fibonacci e restituisce un valore. Il ciclo `for` prende questo valore e lo assegna a `n`, poi esegue il corpo del ciclo `for` per quel valore di `n`.
- Come fa il ciclo `for` a sapere quando fermarsi? Sono contento che lo abbiate chiesto! Quando `next(fib_iter)` solleva un'eccezione di tipo `StopIteration`, il ciclo `for` sopprimerà l'eccezione e uscirà

normalmente. (Ogni altra eccezione vi passerà attraverso e verrà sollevata come al solito.) E dov'è che avete visto un'eccezione di tipo `StopIteration`? Nel metodo `__next__()`, naturalmente!



7.6. UN ITERATORE DI REGOLE PER I SOSTANTIVI PLURALI

Ora è il momento del gran finale. Riscriviamo il generatore di regole per i sostantivi plurali sotto forma di iteratore.

iter(f) chiama

f.__iter__

next(f)

chiama

f.__next__

```

class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename, encoding='utf-8')
        self.cache = []

    def __iter__(self):
        self.cache_index = 0
        return self

    def __next__(self):
        self.cache_index += 1
        if len(self.cache) >= self.cache_index:
            return self.cache[self.cache_index - 1]

        if self.pattern_file.closed:
            raise StopIteration

        line = self.pattern_file.readline()
        if not line:
            self.pattern_file.close()
            raise StopIteration

        pattern, search, replace = line.split(None, 3)
        funcs = build_match_and_apply_functions(
            pattern, search, replace)
        self.cache.append(funcs)
        return funcs

rules = LazyRules()

```

Così questa classe implementa `__iter__()` e `__next__()` e perciò può essere usata come un iteratore. Dunque, ne create un'istanza e l'assegnate a `rules`. Questo accade solo una volta, al momento di importare il modulo.

Analizziamo la classe un pezzo alla volta.

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename, encoding='utf-8') ①
        self.cache = [] ②
```

1. Quando istanziamo la classe `LazyRules`, apriamo il file dei pattern ma non lo leggiamo. (Lo faremo più tardi.)
2. Dopo aver aperto il file dei pattern, inizializzate la cache che userete più tardi (nel metodo `__next__()`) durante la lettura dei pattern dal file.

Prima di continuare, diamo un'occhiata più da vicino a `rules_filename`. Non è definita all'interno del metodo `__iter__()`. In effetti, non è definita all'interno di *nessun* metodo. È definita a livello della classe. È una *variabile di classe*, e sebbene possiate accedervi esattamente come una variabile di istanza (`self.rules_filename`), è condivisa da tutte le istanze della classe `LazyRules`.

```

>>> import plural6
>>> r1 = plural6.LazyRules()
>>> r2 = plural6.LazyRules()
>>> r1.rules_filename ①
'plural6-rules.txt'
>>> r2.rules_filename
'plural6-rules.txt'
>>> r2.rules_filename = 'r2-override.txt' ②
>>> r2.rules_filename
'r2-override.txt'
>>> r1.rules_filename
'plural6-rules.txt'
>>> r2.__class__.rules_filename ③
'plural6-rules.txt'
>>> r2.__class__.rules_filename = 'papayawhip.txt' ④
>>> r1.rules_filename
'papayawhip.txt'
>>> r2.rules_filename ⑤
'r2-overridetxt'

```

1. Ogni istanza della classe eredita l'attributo `rules_filename` con il valore definito dalla classe.
2. Cambiare il valore dell'attributo in un'istanza non ha effetto sulle altre istanze...
3. ...né modifica l'attributo di classe. Potete accedere all'attributo di classe (al contrario dell'attributo di una singola istanza) usando lo speciale attributo `__class__` per accedere alla classe stessa.
4. Se modificate l'attributo di classe, tutte le istanze che stanno ancora ereditando quel valore (come `r1` qui) verranno affette dal cambiamento.
5. Le istanze che hanno sostituito quell'attributo (come `r2` qui) non verranno affette.

E ora torniamo al nostro spettacolo.

```

def __iter__(self): ①
    self.cache_index = 0
    return self ②

```

1. Il metodo `__iter__()` verrà chiamato ogni volta che qualcuno — diciamo, un ciclo `for` — chiama `iter(rules)`.
2. L'unica cosa che ogni metodo `__iter__()` deve fare è restituire un iteratore. In questo caso, il metodo restituisce `self`, segnalando che questa classe definisce un metodo `__next__()` che si prenderà cura di restituire i valori durante l'iterazione.

```
def __next__(self):①  
    .  
    .  
    .  
    pattern, search, replace = line.split(None, 3)  
    funcs = build_match_and_apply_functions(②  
        pattern, search, replace)  
    self.cache.append(funcs)③  
    return funcs
```

1. Il metodo `__next__()` viene chiamato ogni volta che qualcuno — diciamo, un ciclo `for` — chiama `next(rules)`. Questo metodo si capisce meglio se cominciamo dalla fine e lo leggiamo all'indietro. Quindi, facciamolo.
2. L'ultima parte di questa funzione dovrebbe sembrare quantomeno familiare. La funzione `build_match_and_apply_functions()` non è cambiata, è la stessa che è sempre stata.
3. L'unica differenza è che, prima di restituire le funzioni di ricerca e sostituzione (che sono memorizzate nella tupla `funcs`), dobbiamo salvarle in `self.cache`.

Muovendoci all'indietro...

```

def __next__(self):
    .
    .
    .
    line = self.pattern_file.readline() ①
    if not line: ②
        self.pattern_file.close()
        raise StopIteration ③
    .
    .
    .

```

1. Qui utilizziamo i file in maniera astuta. Il metodo `readline()` (notate: singolare, non il plurale `readlines()`) legge esattamente una riga da un file aperto. Nello specifico, la riga successiva. *(Anche gli oggetti file sono iteratori! Ci sono iteratori dappertutto...)*
2. Se `readline()` legge una riga, `line` non sarà una stringa vuota. Anche se il file contenesse una riga vuota, `line` finirebbe per essere la stringa `'\n'` contenente il carattere di ritorno a capo. Quando `line` è davvero la stringa vuota significa che non ci sono più righe da leggere nel file.
3. Quando raggiungiamo la fine del file, dobbiamo chiudere il file e lanciare la magica eccezione di tipo `StopIteration`. Ricordate, siamo arrivati a questo punto perché avevamo bisogno di una coppia di funzioni di ricerca e sostituzione per la regola successiva. Tale regola proviene dalla riga successiva nel file... ma non c'è nessun'altra riga! Di conseguenza, non abbiamo alcun valore da restituire. L'iterazione è finita. (🎵 The party's over... 🎵)

Spostandoci all'indietro fino all'inizio del metodo `__next__()`...

```

def __next__(self):
    self.cache_index += 1
    if len(self.cache) >= self.cache_index:
        return self.cache[self.cache_index - 1]    ①

    if self.pattern_file.closed:
        raise StopIteration    ②

    .
    .
    .

```

1. `self.cache` sarà una lista di funzioni per cercare corrispondenze con le singole regole e poi sostituirle. (Almeno *questo* dovrebbe suonarvi familiare!) `self.cache_index` tiene traccia di quale elemento in cache dovremo restituire successivamente. Se non abbiamo ancora esaurito la cache (cioè se la lunghezza di `self.cache` è maggiore di `self.cache_index`), allora l'accesso alla cache ha successo! Urrà! Possiamo restituire le funzioni di ricerca e sostituzione dalla cache anziché costruirle da zero.
2. D'altra parte, se il tentativo di accesso alla cache fallisce e l'oggetto `file` è stato chiuso (cosa che potrebbe accadere più avanti nel metodo, come avete visto nel precedente frammento di codice) allora non c'è nient'altro che possiamo fare. Se il file è chiuso, vuol dire che lo abbiamo esaurito — abbiamo già letto ogni riga dal file dei pattern e abbiamo già costruito e memorizzato le funzioni di ricerca e sostituzione per ogni pattern. Il file è esaurito; la cache è esaurita; anche io sono ormai esaurito... Ma tenete duro, abbiamo quasi finito.

Mettendo insieme tutti i pezzi, ecco cosa succede e quando.

- Quando il modulo viene importato, viene creata una singola istanza della classe `LazyRules`, chiamata `rules`, che apre il file dei pattern ma non lo legge.
- Quando si richiedono a `rules` le prime funzioni di ricerca e sostituzione, l'istanza controlla la propria cache ma scopre che è vuota. Quindi legge una singola riga dal file dei pattern, costruisce le funzioni di ricerca e sostituzione per i pattern di quella riga e le memorizza nella cache.
- Diciamo che, per fare un esempio, proprio la prima regola trova una corrispondenza. Se è così, nessuna nuova coppia di funzioni viene costruita e nessun'altra riga viene letta dal file.

- In più, sempre per fare un esempio, supponiamo che il chiamante invochi *di nuovo* la funzione `plural()` per costruire il plurale di una parola diversa. Il ciclo `for` nella funzione `plural()` chiamerà `iter(rules)`, che azzererà l'indice della cache ma non toccherà l'oggetto file aperto.
- Alla prima iterazione, il ciclo `for` chiederà un valore a `rules`, che invocherà il proprio metodo `__next__()`. Questa volta, però, la cache è caricata con una singola coppia di funzioni di ricerca e sostituzione, corrispondenti ai pattern nella prima riga del file dei pattern. Visto che la coppia è stata costruita e memorizzata nel tentativo di pluralizzare la parola precedente, viene recuperata dalla cache. L'indice della cache viene incrementato e il file aperto non viene mai toccato.
- Diciamo ora che, per fare un esempio, la prima regola questa volta non genera alcuna corrispondenza. Quindi il ciclo `for` ricomincia da capo e chiede un altro valore a `rules`, che invoca il metodo `__next__()` una seconda volta. Questa volta la cache è esaurita — conteneva un solo elemento e noi ne stiamo chiedendo un secondo — quindi il metodo `__next__()` prosegue. Legge un'altra riga dal file aperto, costruisce le funzioni di ricerca e sostituzione sulla base dei pattern e le memorizza in cache.
- Questo processo di lettura-costruzione-memorizzazione continuerà fino a quando le regole lette dal file dei pattern non corrisponderanno alla parola che stiamo cercando di pluralizzare. Se troviamo una regola che corrisponde prima della fine del file, semplicemente la usiamo e ci fermiamo, con il file ancora aperto. Il puntatore del file rimarrà ovunque avremo smesso di leggere, aspettando la prossima invocazione di `readline()`. Nel frattempo, la cache contiene ora più elementi, e se ricominciamo ancora a cercare di pluralizzare una nuova parola ognuno degli elementi in cache verrà provato prima di leggere la riga successiva dal file dei pattern.

Abbiamo raggiunto il nirvana della pluralizzazione.

1. **Costo di inizializzazione minimo.** L'unica cosa che accade durante l'importazione del modulo è la creazione di una singola istanza di una classe e l'apertura di un file (ma senza alcuna lettura).
2. **Prestazioni massime.** L'esempio precedente avrebbe letto tutto il file e costruito funzioni dinamicamente ogni volta che volevate pluralizzare una parola. Questa versione memorizza le funzioni non appena vengono costruite e, nel caso peggiore, leggerà tutto il file dei pattern una volta sola a prescindere dal numero di parole che pluralizzate.
3. **Separazione di codice e dati.** Tutti i pattern sono archiviati in un file separato. Il codice è codice, i dati sono dati, “e mai i due si incontreranno”.



Questo è veramente il nirvana? Be', sì e no. Ecco una cosa da considerare a proposito dell'esempio con LazyRules: il file dei pattern aperto durante l'esecuzione di `__init__()` rimane aperto fino a quando viene raggiunta l'ultima regola. Python chiuderà il file alla fine, quando uscite dall'interprete, o dopo che l'ultima istanza della classe LazyRules viene distrutta, ma l'intervallo di tempo trascorso fino a quel momento potrebbe ancora essere molto *lungo*. Se questa classe è parte di un processo Python di lunga durata, l'interprete potrebbe non terminare mai e l'oggetto LazyRules potrebbe non venire mai distrutto.

Ci sono alcuni modi per aggirare questo problema. Invece di aprire il file durante l'esecuzione di `__init__()` e lasciarlo aperto mentre leggete le regole una alla volta, potreste aprire il file, leggere tutte le regole e chiudere immediatamente il file. Oppure potreste aprire il file, leggere una regola, salvare la posizione di lettura corrente del file con il metodo `tell()`, chiudere il file e riaprirlo più tardi, usando il metodo `seek()` per continuare a leggere dal punto in cui eravate rimasti. Oppure potreste non preoccuparvi del problema e lasciare semplicemente il file aperto, come in questo esempio di codice. La programmazione è progettazione, e la progettazione è fatta di compromessi e vincoli. Lasciare un file aperto troppo a lungo potrebbe essere un problema; rendere il vostro codice più complicato potrebbe essere un problema. Qual è il problema più grande dipende dal vostro gruppo di sviluppo, dalla vostra applicazione e dal vostro ambiente di esecuzione.



7.7. LETTURE DI APPROFONDIMENTO

- Tipi di iteratore
- PEP 234: Iteratori
- PEP 255: Generatori semplici
- Trucchi con i generatori per programmatori di sistema

CAPITOLO 8. USO AVANZATO DEGLI ITERATORI

“ Le grandi pulci portano piccole pulci che le mordono con appetito,
e le pulci piccole hanno pulci più piccole, e così via all’infinito. ”

— Augustus De Morgan

8.1. IMMERSIONE!

Laddove le espressioni regolari accrescono enormemente le funzionalità delle stringhe, il modulo `itertools` agisce allo stesso modo nei confronti degli iteratori. Ma prima di esaminare le meraviglie di questo modulo, voglio mostrarvi un classico rompicapo.

```
HAWAII + IDAHO + IOWA + OHIO == STATES  
510199 + 98153 + 9301 + 3593 == 621246
```

```
H = 5  
A = 1  
W = 0  
I = 9  
D = 8  
O = 3  
S = 6  
T = 2  
E = 4
```

Rompicapi come questi sono chiamati *criptarismi* o più precisamente criptarismi *alfametici*. Le lettere formano parole vere, ma se sostituite ogni lettera con una cifra da 0 a 9 il rompicapo “forma” anche un’equazione aritmetica. Il trucco è scoprire quali lettere corrispondono a quali cifre. Tutte le occorrenze di ogni lettera devono corrispondere alla stessa cifra, nessuna cifra può essere ripetuta e nessuna “parola” può cominciare con la cifra 0.

In questo capitolo, analizzeremo un incredibile programma Python originariamente scritto da Raymond Hettinger. Questo programma risolve rompicapi di alfabetica *in sole 14 righe di codice*.

*Il rompicapo
alfabetico più
conosciuto è*

SEND +

MORE =

MONEY.

```

import re
import itertools

def solve(puzzle):
    words = re.findall('[A-Z]+', puzzle.upper())
    unique_characters = set(''.join(words))
    assert len(unique_characters) <= 10, 'Troppe lettere'
    first_letters = {word[0] for word in words}
    n = len(first_letters)
    sorted_characters = ''.join(first_letters) + \
        ''.join(unique_characters - first_letters)
    characters = tuple(ord(c) for c in sorted_characters)
    digits = tuple(ord(c) for c in '0123456789')
    zero = digits[0]
    for guess in itertools.permutations(digits, len(characters)):
        if zero not in guess[:n]:
            equation = puzzle.translate(dict(zip(characters, guess)))
            if eval(equation):
                return equation

if __name__ == '__main__':
    import sys
    for puzzle in sys.argv[1:]:
        print(puzzle)
        solution = solve(puzzle)
        if solution:
            print(solution)

```

Potete eseguire il programma dalla riga di comando. Su Linux, avrete qualcosa di simile a questo. (Le seguenti esecuzioni potrebbero impiegare un po' di tempo a seconda della velocità del vostro computer, e non c'è nessun indicatore di progresso. Siate pazienti!)

```

you@localhost:~/diveintopython3/esempi$ python3 alphametics.py "HAWAII + IDAHO + IOWA + OHIO = STATES"
HAWAII + IDAHO + IOWA + OHIO = STATES
510199 + 98153 + 9301 + 3593 == 621246

you@localhost:~/diveintopython3/esempi$ python3 alphametics.py "I + LOVE + YOU == DORA"
I + LOVE + YOU == DORA
1 + 2784 + 975 == 3760

you@localhost:~/diveintopython3/esempi$ python3 alphametics.py "SEND + MORE == MONEY"
SEND + MORE == MONEY
9567 + 1085 == 10652

```



8.2. TROVARE TUTTE LE OCCORRENZE DI UN PATTERN

Come prima cosa, questo risolutore di alfabetica trova tutte le lettere (A–Z) nel rompicapo.

```

>>> import re

>>> re.findall('[0-9]+', '16 pezzi 2x4 in righe di 8') ①
['16', '2', '4', '8']

>>> re.findall('[A-Z]+', 'SEND + MORE == MONEY')      ②
['SEND', 'MORE', 'MONEY']

```

1. Il modulo `re` è l'implementazione Python delle espressioni regolari. Include un'elegante funzione chiamata `findall()` che accetta un pattern di espressione regolare e una stringa e trova tutte le occorrenze del pattern all'interno della stringa. In questo caso il pattern corrisponde a sequenze di numeri. La funzione `findall()` restituisce una lista di tutte le sottostringhe che corrispondono al pattern.
2. Qui il pattern di espressione regolare corrisponde a sequenze di lettere. Ancora una volta, il valore di ritorno è una lista e ogni elemento nella lista è una stringa che corrisponde a quel pattern di espressione regolare.

Ecco un altro esempio per mettere alla prova la vostra mente.

```
>>> re.findall(' c.*? c', 'Chi ama chiama chi ama, chiamami tu che chi ami chiami.')
[' chiama c', ' chiamami tu c', ' chi ami c']
```

Sorpresi? L'espressione regolare cerca uno spazio, una c e poi la serie di caratteri più corta possibile contenente qualsiasi carattere (. * ?), poi uno spazio, poi un'altra c. Bene, guardando la stringa in ingresso, vedo cinque corrispondenze:

1. Chi ama **chiama** chi ama, chiamami tu che chi ami chiami.
2. Chi ama chiama **chi ama**, chiamami tu che chi ami chiami.
3. Chi ama chiama chi ama, **chiamami tu** che chi ami chiami.
4. Chi ama chiama chi ama, chiamami tu **che** chi ami chiami.
5. Chi ama chiama chi ama, chiamami tu che **chi ami** chiami.

Ma la funzione `re.findall()` restituisce solo tre corrispondenze. Nello specifico, restituisce la prima, la terza e la quinta. Come mai? Perché *non restituisce corrispondenze sovrapposte*. La prima corrispondenza si sovrappone alla seconda, quindi la prima viene restituita e la seconda viene saltata. Poi la terza si sovrappone alla quarta, quindi la terza viene restituita e la quarta viene saltata. Infine, la quinta viene restituita. Tre corrispondenze, non cinque.

Questo non ha nulla a che fare con il risolutore di alfabetica; ho solo pensato che fosse interessante.

*Questo è uno
degli
scioglilingua
più facili che
esistono in
italiano.*



8.3. ELIMINARE GLI ELEMENTI RIPETUTI DA UNA SEQUENZA

Gli insiemi rendono banale eliminare gli elementi ripetuti da una sequenza.

```
>>> a_list = ['Chi', 'ama', 'chiama', 'chi', 'ama', 'chiamami', 'tu', 'che', 'ami', 'chi', 'chiami']
>>> set(a_list)                                ①
{'ami', 'che', 'ama', 'Chi', 'tu', 'chi', 'chiamami', 'chiama', 'chiami'}
>>> a_string = 'EAST IS EAST'
>>> set(a_string)                              ②
{'A', ' ', 'E', 'I', 'S', 'T'}
>>> words = ['SEND', 'MORE', 'MONEY']
>>> ''.join(words)                             ③
'SENDMOREMONEY'
>>> set(''.join(words))                        ④
{'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
```

1. Data una lista contenente diverse stringhe, la funzione `set()` restituirà un insieme di stringhe uniche a partire dalla lista. Questa operazione ha senso se la pensate come un ciclo `for`. Prendete il primo elemento della lista, mettetelo nell'insieme. Il secondo. Il terzo. Il quarto. Il quinto — aspettate, questo è già nell'insieme, quindi va elencato solo una volta, perché gli insiemi Python non accettano ripetizioni. Il sesto, il settimo, l'ottavo, il nono. Poi il decimo — anche questo elemento è ripetuto, quindi va elencato solo una volta. E l'undicesimo. Il risultato finale? Un insieme di elementi unici creato a partire dalla lista originale, senza alcuna ripetizione. La lista originale non va nemmeno ordinata prima.
2. La stessa tecnica funziona con le stringhe, dato che una stringa è solamente una sequenza di caratteri.
3. Data una lista di stringhe, `''.join(lista)` le concatena tutte in un'unica stringa.
4. Quindi, data una lista di stringhe, questa riga di codice restituisce un insieme di caratteri unici senza alcuna ripetizione creato a partire da tutte le stringhe.

Il risolutore di alfabetica usa questa tecnica per ottenere un insieme di tutti i caratteri unici nel rompicapo.

```
unique_characters = set(''.join(words))
```

Questo insieme viene successivamente usato per assegnare le cifre ai caratteri man mano che il risolutore itera attraverso le possibili soluzioni.



8.4. FARE ASSERTZIONI

Come molti linguaggi di programmazione, Python possiede un'istruzione `assert`. Ecco come funziona.

```
>>> assert 1 + 1 == 2 ①
>>> assert 1 + 1 == 3 ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert 2 + 2 == 5, "Solo per valori molto grandi di 2" ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Solo per valori molto grandi di 2
```

1. L'istruzione `assert` è seguita da una qualsiasi espressione Python valida. In questo caso l'espressione `1 + 1 == 2` viene valutata a `True`, quindi l'istruzione `assert` non fa nulla.
2. Tuttavia, se l'espressione Python viene valutata a `False`, l'istruzione `assert` solleverà un'eccezione di tipo `AssertionError`.
3. Potete anche includere un messaggio di spiegazione che viene stampato nel caso l'eccezione `AssertionError` venga sollevata.

Perciò, questa riga di codice:

```
assert len(unique_characters) <= 10, 'Troppe lettere'
```

...equivale a queste due righe:

```
if len(unique_characters) > 10:
    raise AssertionError('Troppe lettere')
```

Il risolutore di alfabetica usa proprio questa istruzione `assert` per interrompere la propria esecuzione nel caso il rompicapo contenga più di dieci lettere uniche. Dato che a ogni lettera viene assegnata una singola cifra e che ci sono solo dieci cifre, un rompicapo con più di dieci lettere uniche non può avere una soluzione.



8.5. ESPRESSIONI GENERATORE

Un'espressione generatore è come una funzione generatore senza la funzione.

```
>>> unique_characters = {'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
>>> gen = (ord(c) for c in unique_characters) ①
>>> gen                                         ②
<generator object <genexpr> at 0x00BADC10>
>>> next(gen)                                 ③
69
>>> next(gen)
68
>>> tuple(ord(c) for c in unique_characters) ④
(69, 68, 77, 79, 78, 83, 82, 89)
```

1. Un'espressione generatore è come una funzione anonima che produce valori. L'espressione stessa somiglia a una descrizione di lista, ma è circondata da parentesi tonde invece che parentesi quadre.
2. Un'espressione generatore restituisce... un iteratore.
3. Invocare `next(gen)` restituisce il valore successivo dell'iteratore.
4. Se volete, potete iterare attraverso tutti i possibili valori e restituire una tupla, una lista, o un insieme passando l'espressione generatore a `tuple()`, `list()`, o `set()`. In questi casi non avete bisogno di un insieme aggiuntivo di parentesi — vi basta passare l'espressione `ord(c) for c in unique_characters` “nuda e cruda” alla funzione `tuple()` e Python scoprirà che è un'espressione generatore.

☞ Usare un'espressione generatore invece di una descrizione di lista può risparmiare sia CPU che RAM. Se state costruendo una lista solo per buttarla via (per esempio, passandola a `tuple()` o `set()`), è meglio usare un'espressione generatore!

Ecco un altro modo di realizzare la stessa cosa, utilizzando una funzione generatore:

```
def ord_map(a_string):  
    for c in a_string:  
        yield ord(c)  
  
gen = ord_map(unique_characters)
```

L'espressione generatore è più compatta, ma funzionalmente equivalente.

*
**

8.6. CALCOLARE LE PERMUTAZIONI... IN MANIERA RITARDATA!

Prima di tutto, cosa diavolo sono le permutazioni? Le permutazioni sono un concetto matematico. (Ci sono effettivamente diverse definizioni, a seconda di quale matematica state utilizzando. Qui sto parlando di combinatoria, ma se questo non vi dice niente, non preoccupatevi. Come sempre, Wikipedia è vostra amica.)

L'idea alla base delle permutazioni è quella di prendere una lista di cose (potrebbero essere numeri, potrebbero essere lettere, potrebbero essere orsi danzanti) e trovare tutti i modi possibili per dividerla in liste più piccole. Tutte le liste più piccole hanno la stessa dimensione, che può essere tanto piccola quanto 1 e tanto grande quanto il numero totale degli elementi. Oh, e nessun oggetto può essere ripetuto. I matematici dicono cose come “troviamo le permutazioni di 3 elementi differenti presi 2 alla volta”, che significa trovare tutte le possibili coppie ordinate a partire da una sequenza di 3 elementi.

```

>>> import itertools ①
>>> perms = itertools.permutations([1, 2, 3], 2) ②
>>> next(perms) ③
(1, 2)
>>> next(perms)
(1, 3)
>>> next(perms)
(2, 1) ④
>>> next(perms)
(2, 3)
>>> next(perms)
(3, 1)
>>> next(perms)
(3, 2)
>>> next(perms) ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

1. Il modulo `itertools` contiene i più svariati tipi di roba divertente, compresa una funzione `permutations()` che si sobbarca tutto il duro lavoro della ricerca di permutazioni.
2. La funzione `permutations()` prende una sequenza (qui una lista di tre interi) e un numero, che è il numero di elementi che volete includere in ogni gruppo più piccolo. La funzione restituisce un iteratore, che potete usare in un ciclo `for` o dovunque sia richiesto un oggetto iterabile. Qui attraverserò l'iteratore manualmente per mostrare tutti i valori.
3. La prima permutazione degli elementi di `[1, 2, 3]` presi 2 alla volta è `(1, 2)`.
4. Notate che le permutazioni sono ordinate: `(2, 1)` è diversa da `(1, 2)`.
5. Ecco qua! Quelle che abbiamo visto sono tutte le permutazioni degli elementi di `[1, 2, 3]` presi 2 alla volta. Coppie come `(1, 1)` e `(2, 2)` non compaiono mai, perché contengono ripetizioni e quindi non sono permutazioni valide. Quando non ci sono più permutazioni, l'iteratore solleva un'eccezione di tipo `StopIteration`.

La funzione `permutations()` non deve per forza prendere una lista. Può accettare qualsiasi sequenza — persino una stringa.

```
>>> import itertools
>>> perms = itertools.permutations('ABC', 3) ①
>>> next(perms)
('A', 'B', 'C') ②
>>> next(perms)
('A', 'C', 'B')
>>> next(perms)
('B', 'A', 'C')
>>> next(perms)
('B', 'C', 'A')
>>> next(perms)
('C', 'A', 'B')
>>> next(perms)
('C', 'B', 'A')
>>> next(perms)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

```
>>> list(itertools.permutations('ABC', 3)) ③
[('A', 'B', 'C'), ('A', 'C', 'B'),
 ('B', 'A', 'C'), ('B', 'C', 'A'),
 ('C', 'A', 'B'), ('C', 'B', 'A')]
```

*Il modulo
itertools
contiene i più
svariati tipi
di roba
divertente.*

1. Una stringa è semplicemente una sequenza di caratteri. Allo scopo di trovare permutazioni, la stringa 'ABC' è equivalente alla lista ['A', 'B', 'C'].
2. La prima permutazione dei 3 elementi in ['A', 'B', 'C'] presi 3 alla volta è ('A', 'B', 'C'). Esistono altre cinque permutazioni — gli stessi tre caratteri in ogni ordine possibile.
3. Dato che la funzione `permutations()` restituisce sempre un iteratore, si può facilmente effettuare il debug sulle permutazioni passando l'iteratore alla funzione built-in `list()` per vedere immediatamente tutte le permutazioni.



8.7. ALTRA ROBA DIVERTENTE NEL MODULO `itertools`

```
>>> import itertools

>>> list(itertools.product('ABC', '123')) ①
[('A', '1'), ('A', '2'), ('A', '3'),
 ('B', '1'), ('B', '2'), ('B', '3'),
 ('C', '1'), ('C', '2'), ('C', '3')]

>>> list(itertools.combinations('ABC', 2)) ②
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

1. La funzione `itertools.product()` restituisce un iteratore contenente il prodotto cartesiano di due sequenze.
2. La funzione `itertools.combinations()` restituisce un iteratore contenente tutte le possibili combinazioni di lunghezza data a partire dalla sequenza data. Questa funzione si comporta come la funzione `itertools.permutations()`, a parte il fatto che le combinazioni non includono elementi che siano una copia di altri elementi in ordine differente. Quindi `itertools.permutations('ABC', 2)` restituirà sia `('A', 'B')` che `('B', 'A')` (tra le altre), ma `itertools.combinations('ABC', 2)` non restituirà `('B', 'A')` perché è una copia di `('A', 'B')` con un diverso ordine degli elementi.

```

>>> names = list(open('esempi/favorite-people.txt', encoding='utf-8')) ①
>>> names
['Dora\n', 'Ethan\n', 'Wesley\n', 'John\n', 'Anne\n',
'Mike\n', 'Chris\n', 'Sarah\n', 'Alex\n', 'Lizzie\n']
>>> names = [name.rstrip() for name in names] ②
>>> names
['Dora', 'Ethan', 'Wesley', 'John', 'Anne',
'Mike', 'Chris', 'Sarah', 'Alex', 'Lizzie']
>>> names = sorted(names) ③
>>> names
['Alex', 'Anne', 'Chris', 'Dora', 'Ethan',
'John', 'Lizzie', 'Mike', 'Sarah', 'Wesley']
>>> names = sorted(names, key=len) ④
>>> names
['Alex', 'Anne', 'Dora', 'John', 'Mike',
'Chris', 'Ethan', 'Sarah', 'Lizzie', 'Wesley']

```

1. Questo idiomma restituisce una lista delle righe in un file di testo.
2. Sfortunatamente (per questo esempio), l'idioma `list(open(nomedelfile))` include anche i ritorni a capo alla fine di ogni riga. Questa descrizione di lista usa il metodo `rstrip()` delle stringhe per togliere lo spazio bianco alla fine di ogni riga. (Le stringhe sono anche dotate di un metodo `lstrip()` per togliere lo spazio bianco all'inizio e di un metodo `strip()` che toglie lo spazio bianco sia all'inizio che alla fine.)
3. La funzione `sorted()` prende una lista e la restituisce ordinata. L'ordine predefinito è quello alfabetico.
4. Ma la funzione `sorted()` può anche prendere una funzione come parametro `key` e ordinare in base a essa. In questo caso la funzione di ordinamento è `len()`, quindi questa invocazione ordina sulla base della lunghezza di ogni elemento calcolando `len(ogni_elemento)`. I nomi più corti vengono prima, poi quelli più lunghi, poi quelli ancora più lunghi.

Che cos'ha a che fare tutto questo con il modulo `itertools`? Sono contento che lo abbiate chiesto.

```


...continuando dalla shell interattiva precedente...
>>> import itertools

>>> groups = itertools.groupby(names, len) ①
>>> groups
<itertools.groupby object at 0x00BB20C0>
>>> list(groups)
[(4, <itertools._grouper object at 0x00BA8BF0>),
 (5, <itertools._grouper object at 0x00BB4050>),
 (6, <itertools._grouper object at 0x00BB4030>)]
>>> groups = itertools.groupby(names, len) ②
>>> for name_length, name_iter in groups: ③
...     print('Nomi con {0:d} lettere:'.format(name_length))
...     for name in name_iter:
...         print(name)
...
Nomi con 4 lettere:
Alex
Anne
Dora
John
Mike
Nomi con 5 lettere:
Chris
Ethan
Sarah
Nomi con 6 lettere:
Lizzie
Wesley

```

- I. La funzione `itertools.groupby()` prende una sequenza e una funzione chiave e restituisce un iteratore che genera coppie. Ogni coppia è composta dal risultato dell'applicazione della funzione chiave a ogni elemento (`funzione_chiave(ogni_elemento)`) e da un altro iteratore che contiene tutti gli elementi che condividono quel risultato.

2. L'invocazione della funzione `list()` ha “esaurito” l'iteratore, cioè avete già generato tutti gli elementi nell'iteratore per costruire la lista. Non c'è nessun pulsante di “reset” su un iteratore, perciò non potete semplicemente ricominciare da capo una volta che lo avete esaurito. Se volete attraversarlo di nuovo (diciamo, nel ciclo `for` successivo) dovete chiamare ancora `itertools.groupby()` per creare un nuovo iteratore.
3. In questo esempio, data una lista di nomi *già ordinata per lunghezza*, `itertools.groupby(names, len)` metterà tutti i nomi di 4 lettere in un iteratore, tutti i nomi di 5 lettere in un altro iteratore, e così via. La funzione `groupby()` è completamente generica; potrebbe raggruppare stringhe sulla base della prima lettera, numeri sulla base del loro numero di fattori oppure utilizzando qualsiasi altra funzione chiave che vi possa venire in mente.

 La funzione `itertools.groupby()` funziona solo se la sequenza in ingresso è già ordinata secondo il criterio seguito dalla funzione di raggruppamento. Nell'esempio precedente avete raggruppato una lista di nomi secondo la funzione `len()`. Questa operazione ha funzionato solo perché la lista in ingresso era già ordinata per lunghezza.

State guardando da vicino?

```
>>> list(range(0, 3))
[0, 1, 2]
>>> list(range(10, 13))
[10, 11, 12]
>>> list(itertools.chain(range(0, 3), range(10, 13))) ①
[0, 1, 2, 10, 11, 12]
>>> list(zip(range(0, 3), range(10, 13))) ②
[(0, 10), (1, 11), (2, 12)]
>>> list(zip(range(0, 3), range(10, 14))) ③
[(0, 10), (1, 11), (2, 12)]
>>> list(itertools.zip_longest(range(0, 3), range(10, 14))) ④
[(0, 10), (1, 11), (2, 12), (None, 13)]
```

1. La funzione `itertools.chain()` prende due iteratori e restituisce un iteratore che contiene tutti gli elementi del primo iteratore seguiti da tutti gli elementi del secondo iteratore. (In realtà, può prendere un numero qualsiasi di iteratori e concatenarli tutti nell'ordine in cui sono stati passati alla funzione.)
2. La funzione `zip()` fa qualcosa di prosaico che si rivela estremamente utile: prende un numero qualsiasi di sequenze e restituisce un iteratore che restituisce tuple contenenti il primo elemento di ogni sequenza, poi il secondo elemento di ognuna, poi il terzo, e così via.
3. La funzione `zip()` si ferma alla fine della sequenza più corta. `range(10, 14)` contiene 4 elementi (10, 11, 12 e 13) ma `range(0, 3)` ne contiene solo 3, quindi la funzione `zip()` restituisce un iteratore di 3 elementi.
4. D'altra parte, la funzione `itertools.zip_longest()` si ferma alla fine della sequenza *più lunga*, inserendo il valore `None` per elementi oltre la fine delle sequenze più corte.

Bene, tutto questo è stato molto interessante, ma come si collega al risolutore di alfabetica? Ecco come:

```
>>> characters = ('S', 'M', 'E', 'D', 'O', 'N', 'R', 'Y')
>>> guess = ('1', '2', '0', '3', '4', '5', '6', '7')
>>> tuple(zip(characters, guess)) ①
(('S', '1'), ('M', '2'), ('E', '0'), ('D', '3'),
 ('O', '4'), ('N', '5'), ('R', '6'), ('Y', '7'))
>>> dict(zip(characters, guess)) ②
{'E': '0', 'D': '3', 'M': '2', 'O': '4',
 'N': '5', 'S': '1', 'R': '6', 'Y': '7'}
```

1. Data una lista di lettere e una lista di cifre (ognuna rappresentata qui come una stringa di 1 carattere), la funzione `zip()` creerà coppie di lettere e cifre, in ordine.
2. Perché è così fantastico? Perché si dà il caso che la struttura dati sia esattamente la struttura giusta da passare alla funzione `dict()` per creare un dizionario che usi le lettere come chiavi e le cifre a esse associate come valori. (Questo non è l'unico modo per farlo, naturalmente. Potreste usare una descrizione di dizionario per creare direttamente il dizionario.) Sebbene la rappresentazione stampata del dizionario elenchi le coppie in ordine differente (i dizionari non hanno alcun “ordine” di per sé), potete vedere che ogni lettera è associata con una cifra sulla base dell'ordinamento delle sequenze `characters` e `guess` originali.

Il risolutore di alfabetica usa questa tecnica per creare un dizionario che mette in relazione le lettere nel rompicampo con le cifre nella soluzione, per ogni possibile soluzione.

```

characters = tuple(ord(c) for c in sorted_characters)
digits = tuple(ord(c) for c in '0123456789')
...
for guess in itertools.permutations(digits, len(characters)):
    ...
    equation = puzzle.translate(dict(zip(characters, guess)))

```

Ma cos'è questo metodo `translate()`? Ah, ora state arrivando alla parte *davvero* divertente.



8.8. UN NUOVO TIPO DI MANIPOLAZIONE DI STRINGHE

Le stringhe Python sono dotate di molti metodi. Avete imparato alcuni di quei metodi nel capitolo sulle stringhe: `lower()`, `count()` e `format()`. Ora voglio introdurvi a una tecnica di manipolazione di stringhe molto potente ma poco conosciuta: il metodo `translate()`.

```

>>> translation_table = {ord('A'): ord('0')} ①
>>> translation_table ②
{65: 79}
>>> 'MARK'.translate(translation_table) ③
'MORK'

```

1. Le traduzioni di stringa cominciano con una tabella di traduzione, che è solo un dizionario che mette in relazione un carattere con un altro. In effetti, “carattere” non è il termine giusto — in realtà la tabella di traduzione mette in relazione un *byte* con un altro.
2. Ricordate, i byte in Python 3 sono interi. La funzione `ord()` restituisce il valore ASCII di un carattere, che nel caso delle lettere A–Z è sempre un byte da 65 a 90.
3. Il metodo `translate()` fa scorrere una stringa attraverso la tabella di traduzione passata come argomento, sostituendo tutte le occorrenze delle chiavi della tabella con i valori corrispondenti. In questo caso, “traducendo” MARK in MORK.

Che cos'ha a che fare questo con la soluzione dei rompicapi alfabetici? A quanto pare, tutto.

*Ora state
arrivando alla
parte davvero
divertente.*

```
>>> characters = tuple(ord(c) for c in 'SMEDONRY') ①
>>> characters
(83, 77, 69, 68, 79, 78, 82, 89)
>>> guess = tuple(ord(c) for c in '91570682') ②
>>> guess
(57, 49, 53, 55, 48, 54, 56, 50)
>>> translation_table = dict(zip(characters, guess)) ③
>>> translation_table
{68: 55, 69: 53, 77: 49, 78: 54, 79: 48, 82: 56, 83: 57, 89: 50}
>>> 'SEND + MORE == MONEY'.translate(translation_table) ④
'9567 + 1085 == 10652'
```

1. Usando una espressione generatore, calcoliamo velocemente i valori in byte per ogni carattere in una stringa. `characters` è un esempio dei valori di `sorted_characters` nella funzione `alphametics.solve()`.
2. Usando un'altra espressione generatore, calcoliamo velocemente i valori in byte per ogni cifra in quella stringa. Il risultato, `guess`, è nella forma restituita dalla funzione `itertools.permutations()` nella funzione `alphametics.solve()`.

3. Questa tabella di traduzione è generata invocando `zip()` su `characters` e `guess` e costruendo un dizionario a partire dalla sequenza di coppie risultante. Questo è esattamente quello che la funzione `alphametics.solve()` fa all'interno del ciclo `for`.
4. Infine, passiamo la tabella di traduzione al metodo `translate()` della stringa del rompicapo originale. Questa operazione converte ogni lettera della stringa nella cifra corrispondente (sulla base delle lettere in `characters` e delle cifre in `guess`). Il risultato è un'espressione Python valida, sotto forma di stringa.

Piuttosto impressionante. Ma cosa potete fare con una stringa che casualmente rappresenta un'espressione Python valida?

*
**

8.9. VALUTARE STRINGHE ARBITRARIE COME ESPRESSIONI PYTHON

Questo è l'ultimo pezzo del puzzle (o piuttosto, l'ultimo pezzo del risolutore di puzzle). Dopo tutta quella sofisticata manipolazione di stringhe, non ci rimane altro che una stringa come `'9567 + 1085 == 10652'`. Ma quella, appunto, è solo una stringa, e a cosa può servirci una stringa? Entra `eval()`, lo strumento universale di valutazione in Python.

```
>>> eval('1 + 1 == 2')
True
>>> eval('1 + 1 == 3')
False
>>> eval('9567 + 1085 == 10652')
True
```

Ma aspettate, c'è di più! La funzione `eval()` non si limita alle espressioni logiche. Può operare su *qualsiasi* espressione Python e restituire *qualsiasi* tipo di dato.

```
>>> eval('"A" + "B"')
'AB'
>>> eval('"MARK".translate({65: 79})')
'MORK'
>>> eval('"AAAAA".count("A"'))
5
>>> eval('["*"] * 5')
['*', '*', '*', '*', '*']
```

Ma aspettate, non è tutto!

```
>>> x = 5
>>> eval("x * 5") ①
25
>>> eval("pow(x, 2)") ②
25
>>> import math
>>> eval("math.sqrt(x)") ③
2.2360679774997898
```

1. L'espressione che `eval()` accetta può fare riferimento a variabili globali definite al di fuori di `eval()`. Se `eval()` viene chiamata all'interno di una funzione, l'espressione può anche fare riferimento a variabili locali.
2. E a funzioni.
3. E a moduli.

Ehi, aspettate un momento...

```
>>> import subprocess
>>> eval("subprocess.getoutput('ls ~')") ①
'Desktop      Library      Pictures \
Documents     Movies       Public   \
Music         Sites'
>>> eval("subprocess.getoutput('rm /un/file/a/caso')") ②
```

1. Il modulo `subprocess` vi permette di eseguire comandi di shell arbitrari e ottenerne il risultato sotto forma di stringa Python.
2. Comandi di shell arbitrari possono avere conseguenze permanenti.

È anche peggio di così, perché esiste una funzione globale chiamata `__import__()` che prende il nome di un modulo sotto forma di stringa, importa il modulo e ne restituisce un riferimento. Combinandola con la potenza di `eval()`, potete costruire una singola espressione che cancellerà tutti i vostri file:

```
>>> eval("__import__('subprocess').getoutput('rm /un/file/a/caso')") ①
```

1. Ora immaginate l'uscita di `'rm -rf ~'`. In realtà non ci sarebbe nessuna uscita, ma d'altra parte non vi sarebbe rimasto più alcun file.

**eval() è
un
PERICOLO**

Be', la parte pericolosa è la valutazione di espressioni arbitrarie provenienti da sorgenti non affidabili. Dovreste usare `eval()` solo su ingressi affidabili. Naturalmente il trucco è scoprire cos'è "affidabile". Ma ecco qualcosa che so di sicuro: **NON** dovreste prendere questo risolutore di alfabetica e metterlo su Internet come un piccolo servizio web divertente. Non fate l'errore di pensare: "Gosh, la funzione effettua un sacco di manipolazioni di stringhe prima di ottenere una stringa da valutare; *non riesco a immaginare* come qualcuno potrebbe servirsene." Qualcuno **SCOPRIRÀ** come infilare codice eseguibile pericoloso attraverso tutte quelle manipolazioni di stringhe (sono accadute cose anche più strane), e poi potrete dare al vostro server il bacio d'addio.

Ma sicuramente ci sarà *qualche* modo per valutare espressioni in sicurezza? Per mettere `eval()` in un ambiente controllato dove non possa accedere al mondo esterno o danneggiarlo? Be', sì e no.

```
>>> x = 5

>>> eval("x * 5", {}, {})                                ①

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'x' is not defined

>>> eval("x * 5", {"x": x}, {})                            ②

>>> import math

>>> eval("math.sqrt(x)", {"x": x}, {})                    ③

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'math' is not defined
```

1. Il secondo e terzo parametro passati alla funzione `eval()` agiscono come spazi di nomi globali e locali in cui valutare l'espressione. In questo caso sono entrambi vuoti, così quando la stringa `"x * 5"` viene valutata non esiste alcun riferimento a `x` né nello spazio di nomi globale né in quello locale, quindi `eval()` solleva un'eccezione.
2. Potete selettivamente includere specifici valori nello spazio di nomi globale elencandoli individualmente. Poi quelle variabili — e solo quelle — saranno disponibili durante la valutazione.
3. Anche se avete appena importato il modulo `math`, non lo avete incluso nello spazio di nomi passato alla funzione `eval()`, così la valutazione è fallita.

Perbacco, è stato facile. Lasciatemi fare quel servizio web di alfabetica ora!

```
>>> eval("pow(5, 2)", {}, {}) ①
25
>>> eval("__import__('math').sqrt(5)", {}, {}) ②
2.2360679774997898
```

1. Anche se avete passato dizionari vuoti per gli spazi di nomi globali e locali, tutte le funzioni built-in di Python sono ancora disponibili durante la valutazione. Quindi `pow(5, 2)` funziona, perché 5 e 2 sono letterali e `pow()` è una funzione built-in.
2. Sfortunatamente (e se non vedete perché è una sfortuna, continuate a leggere), anche la funzione `__import__()` è una funzione built-in e quindi è disponibile.

Sì, questo significa che potete ancora fare cose pericolose, anche se avete esplicitamente impostato gli spazi di nomi globali e locali a dizionari vuoti quando avete invocato `eval()`:

```
>>> eval("__import__('subprocess').getoutput('rm /un/file/a/caso')", {}, {})
```

Oops. Sono contento di non aver fatto quel servizio web di alfabetica. C'è un modo *qualsiasi* di utilizzare `eval()` in sicurezza? Be', sì e no.

```
>>> eval("__import__('math').sqrt(5)",
...      {"__builtins__":None}, {}) ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
>>> eval("__import__('subprocess').getoutput('rm -rf /')",
...      {"__builtins__":None}, {}) ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
```

1. Per valutare espressioni non affidabili in maniera sicura, avete bisogno di definire nello spazio di nomi globale un dizionario che faccia corrispondere `"__builtins__"` a `None`, il valore nullo di Python. Internamente, le funzioni “built-in” sono contenute in uno pseudomodulo chiamato `"__builtins__"`. Questo pseudomodulo (cioè l’insieme delle funzioni built-in) viene reso disponibile alle espressioni valutate a meno che non lo ridefiniate esplicitamente.
2. Assicuratevi di aver ridefinito `__builtins__` e non `__builtin__` o `__built-ins__` o qualche altra variazione che funzionerà benissimo ma vi esporrà a rischi catastrofici.

Quindi `eval()` è sicura ora? Be’, sì e no.

```
>>> eval("2 ** 2147483647",  
...      {"__builtins__":None}, {})
```

 ①

1. Anche senza accedere a `__builtins__` potete ancora lanciare un attacco di negazione del servizio. Per esempio, elevare 2 alla potenza di 2147483647 utilizzerà il 100% della CPU del vostro server per un bel po’ di tempo. (Se state provando a farlo nella shell interattiva, premete alcune volte `Ctrl-C` per interrompere la computazione.) Tecnicamente questa espressione *restituirà* un valore alla fine, ma nel frattempo il vostro server non riuscirà a fare proprio nient’altro.

In conclusione, è possibile valutare in sicurezza espressioni Python non affidabili, per una qualche definizione di “sicurezza” che si scopre non essere terribilmente utile nella vita reale. Va benissimo se state semplicemente divertendovi un po’ e va benissimo se usate solo ingressi affidabili. Ma, in qualsiasi altra situazione, significa andare in cerca di guai.



8.10. METTERE TUTTO INSIEME

Ricapitolando: questo programma risolve rompicapi alfabetici utilizzando la forza bruta, cioè attraverso una ricerca completa di tutte le possibili soluzioni. Per fare questo, il programma...

1. Trova tutte le lettere nel rompicapo con la funzione `re.findall()`.

2. Trova tutte le lettere *uniche* nel rompicapo eliminando le ripetizioni con gli insiemi e la funzione `set()`.
3. Controlla se ci sono più di 10 lettere uniche (cioè se il rompicapo è assolutamente irrisolvibile) con un'istruzione `assert`.
4. Converte le lettere nei loro equivalenti ASCII con un oggetto generatore.
5. Calcola tutte le possibili soluzioni con la funzione `itertools.permutations()`.
6. Converte ogni possibile soluzione in un'espressione Python con il metodo `translate()` delle stringhe.
7. Verifica ogni possibile soluzione valutando l'espressione Python con la funzione `eval()`.
8. Restituisce la prima soluzione che viene valutata a `True`.

...in sole 14 righe di codice.



8.11. LETTURE DI APPROFONDIMENTO

- Il modulo `itertools`
- `itertools` — Funzioni di iterazione per effettuare cicli in maniera efficiente
- Guardate Raymond Hettinger presentare “IA facile con Python” a PyCon 2009
- Ricetta n°576615: risolutore di alfabetica, il risolutore di alfabetica originale scritto da Raymond Hettinger per Python 2
- Altre ricette di Raymond Hettinger nell'archivio di codice di ActiveState
- Alfabetica su Wikipedia
- Indice dell'alfabetica, compresi moltissimi rompicapi e un generatore per farne dei vostri

Molte grazie a Raymond Hettinger per aver accettato di ridistribuire il suo codice con una licenza diversa in modo che io potessi convertirlo verso Python 3 e usarlo come base per questo capitolo.

CAPITOLO 9. TEST DI UNITÀ

“La convinzione non è la prova della certezza. Siamo stati assolutamente sicuri di molte cose che non erano così.”

— Oliver Wendell Holmes, Jr.

9.1. IMMERSIONE! (OPPURE NO?)

Mentre i ragazzini di oggi vengono rovinati da computer veloci e da stravaganti linguaggi “dinamici” che li inducono a scrivere codice come prima cosa, poi a consegnare il software, e solo dopo a effettuare il debug (sempre che questa attività di correzione degli errori venga svolta), ricordo che ai miei tempi eravamo obbligati a una certa disciplina. Scrivevamo i programmi a *mano*, su *carta*, e il computer li leggeva da *schede perforate*. E tutto questo *ci piaceva!*

In questo capitolo scriverete e farete il debug di un insieme di funzioni di utilità per convertire numeri interi in numeri romani e viceversa. Avete visto le meccaniche per la costruzione e la validazione dei numeri romani nel “Caso di studio: numeri romani”. Ora fate un passo indietro e considerate come potremmo fare per trasformare quel programma in un'utilità a due vie.

Le regole per i numeri romani ci hanno condotto a una serie di interessanti osservazioni.

1. C'è un solo modo corretto di rappresentare un particolare numero come numero romano.
2. Anche l'inverso è vero: se una stringa di caratteri è un numero romano valido, rappresenta un solo numero (cioè, può essere interpretata in un solo modo).
3. C'è un intervallo limitato di numeri esprimibili come numeri romani che nello specifico va da 1 a 3999. I romani avevano diversi modi di esprimere numeri più grandi, per esempio tracciando una barra sopra un numero per dire che il suo valore normale doveva essere moltiplicato per 1000. Per gli scopi di questo capitolo, conveniamo che i numeri romani vadano da 1 a 3999.
4. Non c'è alcun modo di rappresentare 0 nei numeri romani.
5. Non c'è alcun modo di rappresentare numeri negativi nei numeri romani.
6. Non c'è alcun modo di rappresentare frazioni o numeri non interi nei numeri romani.

Cominciamo col tracciare ciò che un modulo `roman.py` dovrebbe fare. Il modulo avrà due funzioni principali, `to_roman()` e `from_roman()`. La funzione `to_roman()` dovrà prendere un intero da 1 a 3999 e restituirne la rappresentazione come numero romano sotto forma di stringa...

Fermiamoci subito qui. Ora facciamo qualcosa di un poco inaspettato: scriviamo un test per controllare che la funzione `to_roman()` faccia quello che volete. Avete letto bene: state per scrivere codice che collauda codice che non avete ancora scritto.

Questa pratica si chiama *sviluppo guidato dai test* (in inglese, *test-driven development* o TDD). L'insieme di due funzioni di conversione — `to_roman()` e più tardi `from_roman()` — può essere implementato e collaudato come una singola unità, separata da qualsiasi programma più grande che le importi. Python possiede un framework per il collaudo di unità, un modulo propriamente chiamato `unittest`.

Il collaudo di unità è una parte importante di una strategia di sviluppo complessivamente centrata sui test. Se scrivete i test di unità, è importante che li scriviate presto e che li manteniate aggiornati man mano che il codice e i requisiti cambiano. Molte persone sostengono che i test vadano scritti prima di scrivere il codice che collaudano e questo è lo stile di sviluppo che mostrerò in questo capitolo. Ma i test di unità portano benefici a prescindere dal momento in cui li scrivete.

- Prima di scrivere codice, scrivere i test di unità vi obbliga a dettagliare i vostri requisiti in modo utile.
- Mentre scrivete codice, i test di unità vi evitano di scriverne troppo. Quando tutti i test passano, la funzione è completa.
- Quando applicate il refactoring al codice, vi assicurano che la nuova versione si comporta allo stesso modo della vecchia versione.
- Durante la manutenzione del codice, i test vi aiutano a pararvi il culo quando qualcuno arriva urlando che il vostro ultimo cambiamento ha guastato il suo vecchio codice (“Ma *signore*, tutti i test di unità passavano quando l’ho inserito nel nostro sistema di controllo di revisione...”)
- Quando scrivete codice in gruppo, avere una serie di test completa riduce drasticamente la probabilità che il vostro codice vada a guastare il codice di qualcun altro, perché prima potete lanciare i loro test di unità. (Ho visto questo tipo di cose nelle maratone di programmazione. Un gruppo si divide i compiti, ognuno prende le specifiche per il proprio compito, scrive i relativi test di unità e poi li condivide con il resto del gruppo. In questo modo, nessuno si spinge talmente lontano fino a sviluppare codice che non si interfaccia bene con quello degli altri.)



9.2. UNA SINGOLA DOMANDA

Un test risponde a una singola domanda sul codice che sta collaudando. Un test dovrebbe essere in grado di...

- ...eseguire completamente da solo, senza alcun intervento umano. Il collaudo di unità ha a che fare con l'automazione.
- ...determinare da solo se la funzione che sta collaudando ha avuto successo oppure è fallita, senza una persona che interpreti i risultati.
- ...eseguire in isolamento, separato da qualsiasi altro test (anche se collaudano la stessa funzione). Ogni test è un'isola.

*Ogni test è
un'isola.*

Detto questo, costruiamo un test per il primo requisito.

- I. La funzione `to_roman()` dovrebbe restituire la rappresentazione come numero romano di tutti gli interi da 1 a 3999.

Non è immediatamente ovvio come questo codice faccia... be', *alcunché*. Definisce una classe che non ha alcun metodo `__init__()`. La classe *ha* un altro metodo, ma non viene mai chiamato. L'intero programma ha un blocco `__main__`, che però non fa riferimento alla classe o al suo metodo. Ma fa qualcosa, prometto.

```
import roman1
import unittest
```

```
class KnownValues(unittest.TestCase):
    known_values = ( (1, 'I'),
                     (2, 'II'),
                     (3, 'III'),
                     (4, 'IV'),
                     (5, 'V'),
                     (6, 'VI'),
                     (7, 'VII'),
                     (8, 'VIII'),
                     (9, 'IX'),
                     (10, 'X'),
                     (50, 'L'),
                     (100, 'C'),
                     (500, 'D'),
                     (1000, 'M'),
                     (31, 'XXXI'),
                     (148, 'CXLVIII'),
                     (294, 'CCXCIV'),
                     (312, 'CCCXII'),
                     (421, 'CDXXI'),
                     (528, 'DXXVIII'),
                     (621, 'DCXXI'),
                     (782, 'DCCLXXXII'),
                     (870, 'DCCCLXX'),
                     (941, 'CMXLI'),
                     (1043, 'MXLIII'),
                     (1110, 'MCX'),
                     (1226, 'MCCXXVI'),
                     (1301, 'MCCCI'),
                     (1485, 'MCDLXXXV'),
                     (1509, 'MDIX'),
```

①

```

(1607, 'MDCVII'),
(1754, 'MDCCCLIV'),
(1832, 'MDCCCXXXII'),
(1993, 'MCMXCIII'),
(2074, 'MMLXXIV'),
(2152, 'MMCLII'),
(2212, 'MMCCXII'),
(2343, 'MMCCCXLIII'),
(2499, 'MMCDXCIX'),
(2574, 'MMDLXXIV'),
(2646, 'MMDCLVI'),
(2723, 'MMDCCXXIII'),
(2892, 'MMDCCCXCII'),
(2975, 'MMCMLXXV'),
(3051, 'MMMLI'),
(3185, 'MMMCLXXXV'),
(3250, 'MMMCCCL'),
(3313, 'MMMCCCXIII'),
(3408, 'MMMCDVIII'),
(3501, 'MMMCI'),
(3610, 'MMMDCX'),
(3743, 'MMMDCCXLIII'),
(3844, 'MMMDCCCXLIV'),
(3888, 'MMMDCCCLXXXVIII'),
(3940, 'MMMCMXL'),
(3999, 'MMMCMXCIX')) ②

```

```

def test_to_roman_known_values(self): ③
    '''to_roman dovrebbe dare un risultato noto con un ingresso noto'''
    for integer, numeral in self.known_values:
        result = roman1.to_roman(integer) ④
        self.assertEqual(numeral, result) ⑤

```



```
if __name__ == '__main__':  
    unittest.main()
```

1. Per scrivere un test, come prima cosa estendete la classe `TestCase` del modulo `unittest`. Questa classe mette a disposizione molti metodi utili che potete usare nel vostro test per esprimere specifiche condizioni di collaudo.
2. Questa è una lista di coppie interi/romani che ho verificato manualmente. Include i dieci numeri più bassi, il numero più alto, ogni numero che si traduce in un numero romano con un singolo carattere e una serie di esempi casuali di altri numeri validi. Non dovete collaudare ogni possibile ingresso, ma dovrete provare a collaudare tutti gli ovvi casi limite.
3. Ogni test individuale ha il proprio metodo. Un metodo di test non prende parametri, non restituisce alcun valore e deve avere un nome che comincia con le quattro lettere `test`. Se il metodo di test ritorna normalmente senza sollevare un'eccezione, il test si considera passato; se il metodo solleva un'eccezione, il test si considera fallito.
4. Qui potete chiamare la funzione `to_roman()`. (Be', la funzione non è ancora stata scritta, ma una volta che lo sarà questa è la riga che la chiamerà.) Notate che avete definito la API per la funzione `to_roman()`: deve prendere un intero (il numero da convertire) e restituire una stringa (la rappresentazione come numero romano). Se la API è differente, il test si considera fallito. Notate anche che state intenzionalmente evitando di catturare alcuna eccezione quando chiamate `to_roman()`. La funzione non dovrebbe sollevare alcuna eccezione quando la chiamate con un ingresso valido, e questi valori di ingresso sono tutti validi. Se `to_roman()` solleva un'eccezione, questo test si considera fallito.
5. Supponendo che la funzione `to_roman()` sia stata definita correttamente, invocata correttamente, abbia completato con successo la propria esecuzione e abbia restituito un valore, l'ultimo passo è controllare se ha restituito il valore *corretto*. Questa è una necessità talmente comune che la classe `TestCase` vi fornisce il metodo `assertEqual()` per controllare se due valori sono uguali. Se il risultato restituito da `to_roman()` (`result`) non corrisponde al valore noto che vi stavate aspettando (`numeral`), `assertEqual()` solleverà un'eccezione e il test fallirà. Se i due valori sono uguali, `assertEqual()` non farà nulla. Se tutti i valori restituiti da `to_roman()` corrispondono ai valori che vi aspettate, `assertEqual()` non solleverà mai alcuna eccezione, così `test_to_roman_known_values()` concluderà normalmente la propria esecuzione, il che significa che `to_roman()` avrà passato questo test.

Ora che avete scritto un test, potete cominciare a implementare la funzione `to_roman()`. Prima di tutto, dovrete creare lo scheletro vuoto della funzione e verificare che il test fallisca. Se il test ha successo prima che abbiate scritto il codice della funzione, i vostri test non stanno collaudando il vostro codice proprio per niente! Il collaudo di unità è un ballo: i test guidano, il codice segue. Scrivete un test che fallisce, poi programmate fino a quando ha successo.

```
# roman1.py

def to_roman(n):
    '''converte un intero in un numero romano'''
    pass
```

①

- I. A questo punto, volete definire la API della funzione `to_roman()`, ma non volete ancora scriverne il codice. (Il vostro test deve prima fallire.) Per creare lo scheletro della funzione, usate la parola riservata `pass` di Python, che non fa proprio nulla.

Per lanciare i test, eseguite `romantest1.py` dalla linea di comando. Se chiamate lo script con l'opzione `-v`, le informazioni visualizzate saranno più verbose in modo che possiate vedere esattamente cosa succede quando ogni test viene eseguito. Con ogni probabilità, il risultato della vostra esecuzione dovrebbe somigliare a questo:

*Scrivete un
test che
fallisce, poi
programmate
fino a quando
ha successo.*

```

you@localhost:~/diveintopython3/esempi$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues) ①
to_roman dovrebbe dare un risultato noto con un ingresso noto ... FAIL ②

=====
FAIL: to_roman dovrebbe dare un risultato noto con un ingresso noto
-----

Traceback (most recent call last):
  File "romantest1.py", line 73, in test_to_roman_known_values
    self.assertEqual(numeral, result)
AssertionError: 'I' != None ③

-----

Ran 1 test in 0.016s ④

FAILED (failures=1) ⑤

```

1. Lanciare lo script esegue `unittest.main()` che a sua volta esegue tutti i test. Ogni test è un metodo all'interno di una classe in `romantest.py`. Queste classi di test non devono essere obbligatoriamente organizzate in alcun modo: possono tutte contenere un singolo metodo di test, oppure potete avere una classe che contiene più metodi di test. L'unico requisito è che ogni classe di test erediti da `unittest.TestCase`.
2. Per ogni test, il modulo `unittest` mostrerà la docstring del metodo e un risultato di successo o fallimento per quel test. Come ci aspettavamo, questo test fallisce.
3. Per ogni test fallito, `unittest` visualizzerà le informazioni della traccia dello stack di esecuzione che mostrano esattamente cos'è successo. In questo caso, l'invocazione di `assertEqual()` ha sollevato un'eccezione di tipo `AssertionError` perché si aspettava che `to_roman(1)` restituisse `'I'`, ma non lo ha fatto. (Dato che non c'era nessuna istruzione esplicita di `return`, la funzione ha restituito `None`, il valore nullo di Python.)
4. Dopo i dettagli di ogni test, `unittest` visualizza un riepilogo di quanti test sono stati eseguiti e quanto tempo hanno impiegato.
5. Nell'insieme, l'esecuzione del collaudo è fallita perché almeno un test non è passato. Quando un test non passa, `unittest` distingue tra fallimenti ed errori. Un fallimento corrisponde all'invocazione di un metodo di asserzione, come `assertEqual()` o `assertRaises()`, che fallisce perché la condizione asserita non è vera o

l'eccezione attesa non è stata sollevata. Un errore corrisponde a qualsiasi altro tipo di eccezione sollevata dal codice che state collaudando o dal test di unità stesso.

Ora finalmente potete scrivere la funzione `to_roman()`.

```
roman_numeral_map = (('M', 1000),
                     ('CM', 900),
                     ('D', 500),
                     ('CD', 400),
                     ('C', 100),
                     ('XC', 90),
                     ('L', 50),
                     ('XL', 40),
                     ('X', 10),
                     ('IX', 9),
                     ('V', 5),
                     ('IV', 4),
                     ('I', 1))
```

①

```
def to_roman(n):
    '''converte un intero in un numero romano'''
    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

②

- I. `roman_numeral_map` è una tupla di tuple che definisce tre cose: la rappresentazione a caratteri dei numeri romani di base; l'ordine dei numeri romani (discendente secondo il valore, da M fino a I); il valore di ogni numero romano. Ogni tupla interna rappresenta una coppia (numero, valore). Non ci sono solo i numeri romani con un singolo carattere; sono definite anche coppie di due caratteri come CM (“un centinaio meno un migliaio”). Questo rende più semplice il codice della funzione `to_roman()`.

2. Ed ecco dove la ricca struttura dati di `roman_numeral_map` vi ricompensa, perché non avete bisogno di alcuna logica speciale per gestire la regola di sottrazione. Per convertire in numeri romani, dovete semplicemente iterare attraverso `roman_numeral_map` cercando il più grande valore intero minore o uguale al numero in ingresso. Una volta trovato, aggiungete la rappresentazione del numero romano alla fine del risultato, sottraete il corrispondente valore intero dal numero in ingresso, insaponate, sciacquate, ripetete.

Se non vi è ancora chiaro come lavora la funzione `to_roman()`, aggiungete una chiamata a `print()` alla fine del ciclo `while`:

```
while n >= integer:
    result += numeral
    n -= integer
    print('sottraggo {0} dal numero in ingresso, aggiungo {1} al risultato'.format(integer, numeral))
```

Con l'istruzione `print()` di debug, il risultato somiglia al seguente:

```
>>> import roman1
>>> roman1.to_roman(1424)
sottraggo 1000 dal numero in ingresso, aggiungo M al risultato
sottraggo 400 dal numero in ingresso, aggiungo CD al risultato
sottraggo 10 dal numero in ingresso, aggiungo X al risultato
sottraggo 10 dal numero in ingresso, aggiungo X al risultato
sottraggo 4 dal numero in ingresso, aggiungo IV al risultato
'MCDXXIV'
```

Quindi la funzione `to_roman()` sembra funzionare, almeno in questo controllo manuale. Ma passerà il test che avete scritto?

```
you@localhost:~/diveintopython3/esempi$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok ①

-----

Ran 1 test in 0.016s

OK
```

- I. Urrà! La funzione `to_roman()` passa il test per i “valori noti”. Certo il test non è completo, ma mette alla prova la funzione con una varietà di ingressi, compresi interi che producono ogni numero romano con un singolo carattere, l’intero più grande possibile (3999) e l’intero che produce il numero romano più lungo (3888). A questo punto, potete essere ragionevolmente fiduciosi che la funzione converta con successo ogni ingresso valido che potreste darle.

Ingresso “valido”? Hmm. E se l’ingresso non fosse valido?

9.3. “FERMATI E PRENDI FUOCO”

Non è sufficiente verificare che una funzione abbia successo quando le viene dato un ingresso valido; dovete anche verificare che fallisca quando le viene dato un ingresso non valido. E non un qualsiasi tipo di fallimento; la funzione deve fallire nel modo che vi aspettate.

*Il modo che
Python ha di*

```
>>> import roman1
>>> roman1.to_roman(4000)
'MMMM'
>>> roman1.to_roman(5000)
'MMMMM'
>>> roman1.to_roman(9000) ①
'MMMMMMMMM'
```

*fermarsi e
prendere
fuoco è
sollevare
un'eccezione.*

- I. Questo non è assolutamente ciò che desiderate — quello non è nemmeno un numero romano valido! In effetti, ognuno di questi numeri è fuori dall'intervallo degli ingressi accettabili, ma la funzione restituisce comunque un finto valore. Restituire silenziosamente valori sbagliati è *maaaaaaale*; se un programma deve fallire, è molto meglio che fallisca velocemente e rumorosamente. “Fermati e prendi fuoco”, come si dice in gergo. Il modo che Python ha di fermarsi e prendere fuoco è sollevare un'eccezione.

La domanda da porsi è: “Come posso esprimere questa idea sotto forma di un requisito collaudabile?” Che ne dite di questo, per cominciare?

La funzione `to_roman()` dovrebbe sollevare un'eccezione di tipo `OutOfRangeError` quando le viene passato un intero più grande di 3999.

Che aspetto potrebbe avere il test?

```
class ToRomanBadInput(unittest.TestCase): ①
    def test_too_large(self): ②
        '''to_roman dovrebbe fallire con numeri grandi'''
        self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000) ③
```

- I. Come nel test precedente, create una classe che eredita da `unittest.TestCase`. Potete avere più di un test per classe (come vedrete più avanti in questo capitolo), ma in questo caso ho scelto di creare una nuova

classe perché questo collaudo è qualcosa di diverso rispetto al precedente. Raggrupperemo tutti i collaudi sugli ingressi validi in una classe e tutti i collaudi sugli ingressi non validi in un'altra.

2. Come nel test precedente, il test vero e proprio è un metodo della classe, con un nome che comincia con `test`.
3. La classe `unittest.TestCase` fornisce il metodo `assertRaises()`, che prende i seguenti argomenti: l'eccezione che vi aspettate, la funzione che state collaudando e gli argomenti che volete passare a quella funzione. (Se la funzione che state collaudando prende più di un argomento, passateli tutti ad `assertRaises()`, in ordine, ed essi verranno passati direttamente alla funzione che state collaudando.)

Fate molta attenzione a quell'ultima riga di codice. Invece di invocare direttamente `to_roman()` e controllare manualmente che sollevi una particolare eccezione (incorporando la chiamata di funzione in un blocco `try...except`), state utilizzando il metodo `assertRaises()`, che ha incapsulato per noi tutte queste operazioni. Tutto quello che dovete fare è dirgli quali sono l'eccezione che vi aspettate (`roman2.OutOfRangeError`), la funzione da invocare (`to_roman()`) e gli argomenti con cui invocarla (`4000`). Il metodo `assertRaises()` si prende cura di chiamare `to_roman()` e di controllare che sollevi `roman2.OutOfRangeError`.

Notate anche che state passando la funzione `to_roman()` stessa come un argomento; non la state invocando e non ne state passando il nome sotto forma di stringa. Ho recentemente menzionato quant'è comodo che ogni cosa in Python sia un oggetto?

Quindi cosa succede quando lanciate la serie di test con questo nuovo test?


```

you@localhost:~/diveintopython3/esempi$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri grandi ... ERROR ①

=====
ERROR: to_roman dovrebbe fallire con numeri grandi
-----

Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AttributeError: 'module' object has no attribute 'OutOfRangeError' ②

-----

Ran 2 tests in 0.000s

FAILED (errors=1)

```

1. Avreste dovuto aspettarvi che questo test fallisse (dato che non avete scritto ancora il codice per farlo passare) ma... non è esattamente “fallito”, bensì c’è stato un “errore”. Questa è una distinzione sottile ma importante. Un test di unità ha effettivamente *tre* valori di ritorno: successo, fallimento ed errore. Successo, ovviamente, significa che il test è passato — il codice ha fatto quello che vi aspettavate. “Fallimento” è il risultato del test precedente (fino a quando non avete scritto il codice per farlo passare) — il codice veniva eseguito ma il risultato non era quello che vi aspettavate. “Errore” significa che il codice non è nemmeno stato eseguito correttamente.
2. Perché il codice non è stato eseguito correttamente? Ce lo dice la traccia dello stack di esecuzione: il modulo che state collaudando non include alcuna eccezione chiamata `OutOfRangeError`. Ricordate, avete passato questa eccezione al metodo `assertRaises()` perché è l’eccezione che volete che la funzione sollevi quando riceve un ingresso fuori dall’intervallo consentito. Ma l’eccezione non esiste, quindi la chiamata al metodo `assertRaises()` fallisce. Il metodo non ha avuto la possibilità di collaudare la funzione `to_roman()`; non è arrivato così lontano.

Per risolvere questo problema, dovete definire l’eccezione `OutOfRangeError` in `roman2.py`.

```
class OutOfRangeError(ValueError): ①
    pass ②
```

1. Le eccezioni sono classi. Un errore di tipo “fuori dall’intervallo” è un tipo di errore sui valori — il valore passato come argomento è fuori dal suo intervallo accettabile. Così questa eccezione eredita dall’eccezione built-in `ValueError`. Questo non è strettamente necessario (avrebbe potuto semplicemente ereditare dalla classe base `Exception`) ma suona corretto.
2. Le eccezioni non fanno effettivamente niente, ma avete bisogno di almeno una riga di codice per costruire una classe. L’istruzione `pass` non fa proprio nulla, ma è una riga di codice Python e quindi costituisce una classe.

Ora lanciate nuovamente la serie di test.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri grandi ... FAIL ①

=====
FAIL: to_roman dovrebbe fallire con numeri grandi
-----

Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AssertionError: OutOfRangeError not raised by to_roman ②

-----

Ran 2 tests in 0.016s

FAILED (failures=1)
```

1. Il nuovo test non è ancora passato, ma non restituisce nemmeno un errore. Invece, il test fallisce. Questo è un passo avanti! Significa che la chiamata al metodo `assertRaises()` questa volta ha avuto successo e che il framework per il collaudo di unità ha effettivamente collaudato la funzione `to_roman()`.
2. Naturalmente, la funzione `to_roman()` non ha sollevato l'eccezione `OutOfRangeError` che avete appena definito, perché non gli avete ancora detto di farlo. Questa è una notizia eccellente! Significa che questo è un test valido — fallisce prima che scriviate il codice per farlo passare.

Ora potete scrivere il codice per far passare questo test.

```
def to_roman(n):
    '''converte un intero in un numero romano'''
    if n > 3999:
        raise OutOfRangeError("numero fuori dall'intervallo (deve essere minore di 4000)") ①

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

1. Questo è semplice: se l'ingresso dato (`n`) è più grande di 3999, allora solleva un'eccezione di tipo `OutOfRangeException`. Il test di unità non controlla il messaggio nella stringa che accompagna l'eccezione, sebbene possiate scrivere un altro test che faccia questo controllo (ma fate attenzione ai problemi di internazionalizzazione per stringhe che variano con l'ambiente o la lingua dell'utente).

Questo fa passare il test? Scopriamolo.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri grandi ... ok
```

①

```
-----
Ran 2 tests in 0.000s
```

OK

- I. Urrà! Entrambi i test passano. Dato che avete lavorato iterativamente, spostandovi avanti e indietro tra i test e il codice, potete essere sicuri che le due righe di codice che avete scritto sono la causa del passaggio di quel test dal “fallimento” al “successo”. Questo tipo di confidenza si guadagna a un certo prezzo, ma si ripagherà da sola durante la vita del vostro codice.

*
**

9.4. PIÙ FERMATE, PIÙ FUOCO

Insieme al collaudo per numeri troppo grandi, avete anche bisogno di collaudare numeri che sono troppo piccoli. Come abbiamo notato nei nostri requisiti funzionali, i numeri romani non possono esprimere lo 0 o i numeri negativi.

```
>>> import roman2
>>> roman2.to_roman(0)
..
>>> roman2.to_roman(-1)
..
```

Questo non va bene. Aggiungiamo un test per ognuna di queste condizioni.

```

class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman dovrebbe fallire con numeri grandi'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 4000) ①

    def test_zero(self):
        '''to_roman dovrebbe fallire con il numero 0'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0) ②

    def test_negative(self):
        '''to_roman dovrebbe fallire con numeri negativi'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1) ③

```

1. Il metodo `test_too_large()` non è cambiato rispetto al passo precedente. Lo includo qui per mostrare dove trova posto il nuovo codice.
2. Ecco un nuovo test: il metodo `test_zero()`. Come il metodo `test_too_large()`, dice al metodo `assertRaises()` definito in `unittest.TestCase` di chiamare la nostra funzione `to_roman()` con un parametro pari a 0 e di verificare che sollevi l'eccezione appropriata, `OutOfRangeError`.
3. Il metodo `test_negative()` è quasi identico, senonché passa -1 alla funzione `to_roman()`. Se uno di questi nuovi test *non* solleva un'eccezione di tipo `OutOfRangeError` (perché la funzione restituisce un valore, o perché solleva qualche altra eccezione) il collaudo si considera fallito.

Ora verifichiamo che i test falliscano:

```

you@localhost:~/diveintopython3/esempi$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
test_negative (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri negativi ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri grandi ... ok
test_zero (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con il numero 0 ... FAIL

=====
FAIL: to_roman dovrebbe fallire con numeri negativi
-----

Traceback (most recent call last):
  File "romantest3.py", line 86, in test_negative
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)
AssertionError: OutOfRangeError not raised by to_roman

=====
FAIL: to_roman dovrebbe fallire con il numero 0
-----

Traceback (most recent call last):
  File "romantest3.py", line 82, in test_zero
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)
AssertionError: OutOfRangeError not raised by to_roman

-----

Ran 4 tests in 0.000s

FAILED (failures=2)

```

Eccellente. Entrambi i test falliscono, come era da aspettarsi. Ora passiamo al codice e vediamo cosa possiamo fare per farli passare.

```

def to_roman(n):
    '''converte un intero in un numero romano'''
    if not (0 < n < 4000): ①
        raise OutOfRangeError("numero fuori dall'intervallo (deve essere tra 1 e 3999)") ②

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result

```

1. Questa è una classica scorciatoia di Python: molteplici confronti allo stesso tempo. Questo codice è equivalente a `if not ((0 < n) and (n < 4000))`, ma è molto più facile da leggere. Questa riga di codice dovrebbe catturare gli ingressi che sono troppo grandi, negativi, o zero.
2. Se cambiate le vostre condizioni, assicuratevi di aggiornare i vostri messaggi in modo che riflettano i cambiamenti. Il framework `unittest` non se ne curerà, ma sarebbe difficile effettuare un controllo manuale se il vostro codice lancia eccezioni non correttamente descritte.

Potrei fornirvi un'intera serie di esempi non correlati per mostrarvi che la scorciatoia dei molteplici confronti allo stesso tempo funziona, ma invece eseguirò semplicemente i test di unità e lo dimostrerò.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
test_negative (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri negativi ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri grandi ... ok
test_zero (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con il numero 0 ... ok
```

Ran 4 tests in 0.016s

OK

*
**

9.5. E ANCORA UNA COSA...

C'era ancora un requisito funzionale per la conversione dei numeri in numeri romani: gestire i numeri non interi.

```
>>> import roman3
>>> roman3.to_roman(0.5) ①
''
>>> roman3.to_roman(1.0) ②
'I'
```

1. Oh, questo non va bene.
2. Oh, questo è ancora peggio. Entrambi i casi dovrebbero sollevare un'eccezione. Invece, danno risultati fasulli.

Fare il collaudo per i numeri non interi non è difficile. Prima di tutto, definite un'eccezione chiamata `NotIntegerError`.

```
# roman4.py  
class OutOfRangeError(ValueError): pass  
class NotIntegerError(ValueError): pass
```

Poi, scrivete un test che verifichi la generazione dell'eccezione `NotIntegerError`.

```
class ToRomanBadInput(unittest.TestCase):  
    .  
    .  
    .  
    def test_non_integer(self):  
        '''to_roman dovrebbe fallire con numeri non interi'''  
        self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
```

Ora controllate che il test fallisca in maniera appropriata.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
test_negative (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri negativi ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri non interi ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri grandi ... ok
test_zero (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con il numero 0 ... ok
```

```
=====
```

```
FAIL: to_roman dovrebbe fallire con numeri non interi
```

```
-----
```

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

```
  File "romantest4.py", line 90, in test_non_integer
```

```
    self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
```

```
AssertionError: NotIntegerError not raised by to_roman
```

```
-----
```

```
Ran 5 tests in 0.000s
```

```
FAILED (failures=1)
```

Scrivete il codice che fa passare il test.

```

def to_roman(n):
    '''converte un intero in un numero romano'''
    if not (0 < n < 4000):
        raise OutOfRangeError("numero fuori dall'intervallo (deve essere tra 1 e 3999)")
    if not isinstance(n, int):
        raise NotIntegerError('numeri non interi non possono essere convertiti') ①
    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result

```

1. La funzione built-in `isinstance()` verifica che una variabile sia di un particolare tipo (o, tecnicamente, di un qualsiasi tipo discendente).
2. Se l'argomento `n` non è un `int`, sollevate la nostra eccezione `NotIntegerError` appena coniata.

Infine, controllate che il codice faccia effettivamente passare il test.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
test_negative (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri negativi ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri non interi ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con numeri grandi ... ok
test_zero (__main__.ToRomanBadInput)
to_roman dovrebbe fallire con il numero 0 ... ok
```

```
-----
Ran 5 tests in 0.000s
```

OK

La funzione `to_roman()` passa tutti i propri test e non me ne vengono in mente altri, quindi è il momento di passare a `from_roman()`.

*
**

9.6. UNA PIACEVOLE SIMMETRIA

Convertire una stringa da un numero romano in un intero sembra più difficile che convertire un intero in un numero romano. Di certo c'è il problema della validazione. È facile controllare se un intero è più grande di 0, ma è un po' più difficile controllare se una stringa contiene un numero romano valido. Tuttavia abbiamo già costruito un'espressione regolare per controllare i numeri romani, quindi questa parte è fatta.

Questo ci lascia il problema di convertire la stringa. Come vedremo fra un minuto, grazie alla ricca struttura dati che abbiamo definito per correlare i singoli numeri romani ai loro valori interi, il succo della funzione `from_roman()` è tanto semplice quanto quello della funzione `to_roman()`.

Ma prima, i test. Avremo bisogno di un test sui “valori noti” per compiere una verifica a campione sull’accuratezza. La nostra serie di test contiene già una corrispondenza tra valori noti, perciò riutilizziamola.

```
def test_from_roman_known_values(self):
    '''from_roman dovrebbe dare un risultato noto con un ingresso noto'''
    for integer, numeral in self.known_values:
        result = roman5.from_roman(numeral)
        self.assertEqual(integer, result)
```

C’è una piacevole simmetria qui. Le funzioni `to_roman()` e `from_roman()` sono una l’inverso dell’altra. La prima converte interi in stringhe in un formato particolare, la seconda converte stringhe in un formato particolare in interi. In teoria, dovremmo essere in grado di far compiere un “viaggio di andata e ritorno” a un numero passandolo alla funzione `to_roman()` per ottenere una stringa, poi passando quella stringa alla funzione `from_roman()` per ottenere un intero, ritrovandoci infine con lo stesso numero.

`n = from_roman(to_roman(n))` per tutti i valori di `n`

In questo caso, “tutti i valori” significa tutti i numeri compresi nell’intervallo 1..3999, dato che questo è l’intervallo di ingressi validi per la funzione `to_roman()`. Possiamo esprimere questa simmetria in un test che scorre tutti i valori tra 1 e 3999, invoca `to_roman()`, invoca `from_roman()` e controlla che il risultato sia uguale al numero originale.

```
class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n per tutti gli n'''
        for integer in range(1, 4000):
            numeral = roman5.to_roman(integer)
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

Questi nuovi test non sono ancora nemmeno in grado di fallire. Non abbiamo definito alcuna funzione `from_roman()`, quindi solleveranno semplicemente qualche errore.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest5.py
```

```
E.E....
```

```
=====
ERROR: test_from_roman_known_values (__main__.KnownValues)
from_roman dovrebbe dare un risultato noto con un ingresso noto
-----
```

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

```
File "romantest5.py", line 78, in test_from_roman_known_values
```

```
    result = roman5.from_roman(numeral)
```

```
AttributeError: 'module' object has no attribute 'from_roman'
```

```
=====
ERROR: test_roundtrip (__main__.RoundtripCheck)
```

```
from_roman(to_roman(n))==n per tutti gli n
-----
```

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

```
File "romantest5.py", line 103, in test_roundtrip
```

```
    result = roman5.from_roman(numeral)
```

```
AttributeError: 'module' object has no attribute 'from_roman'
```

```
-----
Ran 7 tests in 0.019s
```

```
FAILED (errors=2)
```

Un agile scheletro di funzione risolverà questo problema.

```
# roman5.py
```

```
def from_roman(s):
```

```
    '''converte un numero romano in un intero'''
```

(Ehi, avete notato? Ho definito una funzione con solo una docstring. Questo è legale in Python. In effetti, alcuni programmatori venerano questa possibilità. “Non create scheletri vuoti, documentate!”)

Ora i test effettivamente falliranno.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest5.py
```

```
F.F....
```

```
=====
```

```
FAIL: test_from_roman_known_values (__main__.KnownValues)
```

```
from_roman dovrebbe dare un risultato noto con un ingresso noto
```

```
-----
```

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

```
  File "romantest5.py", line 79, in test_from_roman_known_values
```

```
    self.assertEqual(integer, result)
```

```
AssertionError: 1 != None
```

```
=====
```

```
FAIL: test_roundtrip (__main__.RoundtripCheck)
```

```
from_roman(to_roman(n))==n per tutti gli n
```

```
-----
```

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

```
  File "romantest5.py", line 104, in test_roundtrip
```

```
    self.assertEqual(integer, result)
```

```
AssertionError: 1 != None
```

```
-----
```

```
Ran 7 tests in 0.002s
```

```
FAILED (failures=2)
```

Ora è tempo di scrivere la funzione `from_roman()`.

```
def from_roman(s):
    '''converte un numero romano in un intero'''
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral: ①
            result += integer
            index += len(numeral)
    return result
```

- I. Lo schema qui è lo stesso della funzione `to_roman()`. Iterate attraverso la struttura dati dei numeri romani (una tupla di tuple), ma invece di trovare una corrispondenza con i valori interi più alti quanto più spesso è possibile, trovate una corrispondenza con le stringhe dei caratteri numerici romani “più alti” quanto più spesso è possibile.

Se non vi è chiaro come funziona `from_roman()`, aggiungete una chiamata a `print()` alla fine del ciclo `while`:

```
def from_roman(s):
    '''converte un numero romano in un intero'''
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print('trovato', numeral, 'di lunghezza', len(numeral), ', aggiungo', integer)
```



```
>>> import roman5
>>> roman5.from_roman('MCMLXXII')
trovato M di lunghezza 1, aggiungo 1000
trovato CM di lunghezza 2, aggiungo 900
trovato L di lunghezza 1, aggiungo 50
trovato X di lunghezza 1, aggiungo 10
trovato X di lunghezza 1, aggiungo 10
trovato I di lunghezza 1, aggiungo 1
trovato I di lunghezza1, aggiungo 1
1972
```

Tempo di rieseguire i test.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest5.py
```

```
.....
```

```
-----
```

```
Ran 7 tests in 0.060s
```

```
OK
```

Due notizie eccitanti qui. La prima è che la funzione `from_roman()` converte con successo gli ingressi validi, almeno per tutti i valori noti. La seconda è che anche il test di “andata e ritorno” è passato. Combinato con i test sui valori noti, può farvi sentire ragionevolmente sicuri che entrambe le funzioni `to_roman()` e `from_roman()` convertano correttamente tutti i possibili valori validi. (Questo non è garantito; è teoricamente possibile che `to_roman()` abbia un bug che le fa produrre il numero romano sbagliato per un particolare insieme di ingressi e che `from_roman()` abbia un bug reciproco che produce gli stessi valori interi errati proprio per quell’insieme di numeri romani che `to_roman()` genera in maniera scorretta. A seconda della vostra applicazione e dei vostri requisiti, questa possibilità potrebbe infastidirvi; se è così, scrivete test più completi fino a quando non vi sentite tranquilli.)



9.7. ALTRI INGRESSI NON VALIDI

Ora che la funzione `from_roman()` funziona correttamente con ingressi validi, è il momento di inserire l'ultimo pezzo del puzzle: farla funzionare correttamente con ingressi non validi. Questo significa trovare un modo per esaminare una stringa e determinare se è un numero romano valido. Questa operazione è intrinsecamente più difficile rispetto alla validazione di ingressi numerici compiuta nella funzione `to_roman()`, ma avete uno strumento potente a vostra disposizione: le espressioni regolari. (Se non avete familiarità con le espressioni regolari, questo sarebbe un buon momento per leggere il capitolo sulle espressioni regolari.)

Come avete visto nel Caso di studio: numeri romani, esistono alcune semplici regole per costruire un numero romano usando le lettere M, D, C, L, X, V e I. Rivediamo le regole.

- Talvolta i caratteri sono additivi. I è 1, II è 2, e III è 3. VI è 6 (letteralmente, “5 e 1”), VII è 7, e VIII è 8.
- I caratteri delle potenze di dieci (I, X, C e M) possono essere ripetuti fino a tre volte. A 4, dovete sottrarre dal carattere del quintuplo più alto successivo. Non potete rappresentare 4 come IIII; invece, va rappresentato come IV (“1 meno di 5”). Il numero 40 è scritto come XL (“10 meno di 50”), 41 come XLI, 42 come XLII, 43 come XLIII, e poi 44 come XLIV (“10 meno di 50, poi 1 meno di 5”).
- Talvolta i caratteri sono... l'opposto di additivi. Mettendo certi caratteri prima di altri, sottraete dal valore finale. Per esempio, a 9, dovete sottrarre dal carattere della potenza di dieci più alta successiva: 8 è VIII, ma 9 è IX (“1 meno di 10”), non VIIII (dato che il carattere I non può essere ripetuto quattro volte). Il numero 90 è XC, 900 è CM.
- I caratteri dei quintupli non possono essere ripetuti. Il numero 10 è sempre rappresentato come X, mai come VV. Il numero 100 è sempre C, mai LL.
- I numeri romani vengono sempre letti da sinistra a destra, così l'ordine dei caratteri ha molta importanza. DC è 600; CD è un numero completamente differente (400, “100 meno di 500”). CI è 101; IC non è nemmeno un numero romano valido (perché non potete sottrarre 1 direttamente da 100; dovrete scriverlo come XCIX, cioè “10 meno di 100, poi 1 meno di 10”).

Quindi, un test utile sarebbe quello di assicurarsi che la funzione `from_roman()` fallisca quando le passate una stringa con cifre ripetute troppe volte. Quante siano “troppe” dipende dalla cifra.

```

class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman dovrebbe fallire con cifre ripetute troppe volte'''
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)

```

Un altro test utile sarebbe quello di controllare che certi pattern non vengano ripetuti. Per esempio, IX è 9, ma IXIX non è mai valido.

```

def test_repeated_pairs(self):
    '''from_roman dovrebbe fallire con coppie di cifre ripetute'''
    for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
        self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)

```

Un terzo test potrebbe controllare che le cifre appaiano nell'ordine corretto, dal valore più alto a quello più basso. Per esempio, CL è 150, ma LC non è mai valido perché la cifra per 50 non può mai trovarsi prima della cifra per 100. Questo test include un insieme di antecedenti non validi scelti a caso: I prima di M, V prima di X, e così via.

```

def test_malformed_antecedents(self):
    '''from_roman dovrebbe fallire con antecedenti malformati'''
    for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
              'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
        self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)

```

Ognuno di questi test fa affidamento sul fatto che la funzione `from_roman()` sollevi una nuova eccezione, `InvalidRomanNumeralError`, che non abbiamo ancora definito.

```

# roman6.py
class InvalidRomanNumeralError(ValueError): pass

```

Tutti e tre questi test dovrebbero fallire dato che attualmente la funzione `from_roman()` non effettua alcun controllo di validità. (Se non falliscono adesso, allora cosa diavolo stanno collaudando?)

```
you@localhost:~/diveintopython3/esempi$ python3 romantest6.py
```

```
FFF.....
```

```
=====
```

```
FAIL: test_malformed_antecedents (__main__.FromRomanBadInput)
```

```
from_roman dovrebbe fallire con antecedenti malformati
```

```
-----
```

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

```
  File "romantest6.py", line 113, in test_malformed_antecedents
```

```
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

```
AssertionError: InvalidRomanNumeralError not raised by from_roman
```

```
=====
```

```
FAIL: test_repeated_pairs (__main__.FromRomanBadInput)
```

```
from_roman dovrebbe fallire con coppie di cifre ripetute
```

```
-----
```

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

```
  File "romantest6.py", line 107, in test_repeated_pairs
```

```
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

```
AssertionError: InvalidRomanNumeralError not raised by from_roman
```

```
=====
```

```
FAIL: test_too_many_repeated_numerals (__main__.FromRomanBadInput)
```

```
from_roman dovrebbe fallire con cifre ripetute troppe volte
```

```
-----
```

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

```
  File "romantest6.py", line 102, in test_too_many_repeated_numerals
```

```
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

```
AssertionError: InvalidRomanNumeralError not raised by from_roman
```

```
-----
```

```
Ran 10 tests in 0.058s
```

```
FAILED (failures=3)
```

Molto bene. Ora tutto quello che dobbiamo fare è aggiungere l'espressione regolare per verificare i numeri romani validi alla funzione from_roman().

```
roman_numeral_pattern = re.compile('''
    ^                # inizio della stringa
    M{0,3}           # migliaia - da 0 a 3 M
    (CM|CD|D?C{0,3}) # centinaia - 900 (CM), 400 (CD), 0-300 (da 0 a 3 C),
                        # o 500-800 (D, seguita da 0 fino a 3 C)
    (XC|XL|L?X{0,3}) # decine - 90 (XC), 40 (XL), 0-30 (da 0 a 3 X),
                        # o 50-80 (L, seguita da 0 fino a 3 X)
    (IX|IV|V?I{0,3}) # unità - 9 (IX), 4 (IV), 0-3 (da 0 a 3 I),
                        # o 5-8 (V, seguita da 0 fino a 3 I)
    $                # fine della stringa
''', re.VERBOSE)

def from_roman(s):
    '''converte un numero romano in un intero'''
    if not roman_numeral_pattern.search(s):
        raise InvalidRomanNumeralError('Numero romano non valido: {0}'.format(s))

    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index : index + len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

E rieseguire i test...

```
you@localhost:~/diveintopython3/esempi$ python3 romantest7.py
```

```
.....
```

```
-----
```

```
Ran 10 tests in 0.066s
```

```
OK
```

E il premio anticlimax dell'anno va... alla parola "OK", che viene stampata dal modulo unittest quando tutti i test passano.

CAPITOLO 10. REFACTORING

“Dopo aver suonato una quantità immensa di note e ancora altre note, è la semplicità che emerge come l'autentico sigillo dell'arte.”

— Frédéric Chopin

10.1. IMMERSIONE!

Nonostante facciate del vostro meglio per scrivere test di unità completi, i bug capitano. Cosa intendo quando parlo di “bug”? Un bug è un test che non avete ancora scritto.

```
>>> import roman7
>>> roman7.from_roman('') ①
0
```

- I. Questo è un bug. Una stringa vuota dovrebbe sollevare un'eccezione di tipo `InvalidRomanNumeralError`, esattamente come ogni altra sequenza di caratteri che non rappresenta un numero romano valido.

Dopo aver riprodotto il bug, e prima di correggerlo, dovrete scrivere un test che fallisce, mettendo così il bug in evidenza.

```
class FromRomanBadInput(unittest.TestCase):
    .
    .
    .
    def testBlank(self):
        '''from_roman dovrebbe fallire con una stringa vuota'''
        self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, '') ①
```

- I. Qui le cose sono piuttosto semplici. Chiamate `from_roman()` con una stringa vuota e vi assicurate che sollevi un'eccezione di tipo `InvalidRomanNumeralError`. La parte difficile è stata trovare il bug; ora che lo conoscete, scrivere un test per verificarlo è la parte facile.

Dato che il vostro codice ha un bug e ora avete un test che verifica quel bug, il test fallirà:

```
you@localhost:~/diveintopython3/esempi$ python3 romantest8.py -v
from_roman dovrebbe fallire con una stringa vuota ... FAIL
from_roman dovrebbe fallire con antecedenti malformati ... ok
from_roman dovrebbe fallire con coppie di cifre ripetute ... ok
from_roman dovrebbe fallire con cifre ripetute troppe volte ... ok
from_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
from_roman(to_roman(n))==n per tutti gli n ... ok
to_roman dovrebbe fallire con numeri negativi ... ok
to_roman dovrebbe fallire con numeri non interi ... ok
to_roman dovrebbe fallire con numeri grandi ... ok
to_roman dovrebbe fallire con il numero 0 ... ok

=====
FAIL: from_roman dovrebbe fallire con una stringa vuota
-----
Traceback (most recent call last):
  File "romantest8.py", line 117, in test_blank
    self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, '')
AssertionError: InvalidRomanNumeralError not raised by from_roman

-----

Ran 11 tests in 0.171s

FAILED (failures=1)
```

Adesso potete correggere il bug.


```

def from_roman(s):
    '''converte un numero romano in un intero'''

    if not s: ①
        raise InvalidRomanNumeralError('La stringa in ingresso non può essere vuota')

    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError('Numero romano non valido: {}'.format(s)) ②

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result

```

1. Sono necessarie solo due righe di codice: un controllo esplicito per una stringa vuota e un'istruzione di `raise`.
2. Non penso di aver ancora menzionato questa particolarità da nessuna parte in questo libro, quindi vi serva come ultima lezione sulla formattazione di stringhe. A partire da Python 3.1, potete omettere i numeri quando usate gli indici posizionali in una specifica di formato. Questo vuol dire che, invece di usare la specifica di formato `{0}` per fare riferimento al primo parametro del metodo `format()`, potete semplicemente usare `{}`, che Python completerà con l'indice posizionale corretto. Questo funziona con qualsiasi numero di argomenti: la prima `{}` è `{0}`, la seconda `{}` è `{1}`, e così via.

```

you@localhost:~/diveintopython3/esempi$ python3 romantest8.py -v
from_roman dovrebbe fallire con una stringa vuota ... ok ①
from_roman dovrebbe fallire con antecedenti malformati ... ok
from_roman dovrebbe fallire con coppie di cifre ripetute ... ok
from_roman dovrebbe fallire con cifre ripetute troppe volte ... ok
from_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
from_roman(to_roman(n))==n per tutti gli n ... ok
to_roman dovrebbe fallire con numeri negativi ... ok
to_roman dovrebbe fallire con numeri non interi ... ok
to_roman dovrebbe fallire con numeri grandi ... ok
to_roman dovrebbe fallire con il numero 0 ... ok

-----

Ran 11 tests in 0.156s

OK ②

```

1. Il test per la stringa vuota ora ha successo, quindi il bug è stato corretto.
2. Tutti gli altri test hanno ancora successo, il che significa che la correzione del bug non ha introdotto nuovi errori. Potete smettere di programmare.

Programmare in questo modo non rende più facile la correzione dei bug. Per i bug più semplici (come questo) bastano test semplici; bug complessi richiederanno test complessi. In un processo di sviluppo guidato dai test, potrebbe *sembrare* che ci voglia più tempo per correggere un bug, dato che prima avete bisogno di esprimere la sostanza del bug sotto forma di codice (per scrivere il test) e poi dovete correggere il bug vero e proprio. Successivamente, se il test non ha immediatamente successo, avete bisogno di capire se la correzione era sbagliata o se è il test stesso ad avere un bug. Comunque, nel lungo periodo, lavorare spostandovi continuamente avanti e indietro tra il codice del test e il codice che state collaudando vi ricompenserà, perché rende molto più probabile che i bug vengano corretti al primo tentativo. In più, visto che potete facilmente rieseguire *tutti* i test insieme a quello nuovo, avete molte meno probabilità di guastare il vecchio codice quando state correggendo quello nuovo. Il test di unità di oggi è il test di regressione di domani.



10.2. GESTIRE IL CAMBIAMENTO DEI REQUISITI

Nonostante facciate del vostro meglio per tenere i vostri committenti con i piedi ben piantati per terra in modo da ottenere requisiti precisi sotto la minaccia di orribili torture a base di forbici e cera calda, i requisiti cambieranno. La maggior parte dei committenti non sa cosa vuole fino a quando non la vede, e anche se lo sapesse non sarebbe così brava a descrivere ciò che vuole in maniera sufficientemente precisa da risultare utile. E anche se lo fosse, vorrebbe comunque di più nella versione successiva. Quindi siate preparati ad aggiornare i vostri test man mano che i requisiti cambiano.

Supponete per esempio di volere espandere l'intervallo dei numeri romani gestito dalle funzioni di conversione. Normalmente nessun carattere in un numero romano può essere ripetuto più di tre volte di seguito. Ma i Romani erano disposti a fare uno strappo a quella regola per poter rappresentare 4000 con 4 M di seguito. Se fate questa modifica sarete in grado di allargare l'intervallo di numeri convertibili da 1..3999 a 1..4999. Ma prima è necessario fare alcuni cambiamenti ai vostri test.

```

class KnownValues(unittest.TestCase):
    known_values = ( (1, 'I'),
                      .
                      .
                      .
                      (3999, 'MMMCMXCIX'),
                      (4000, 'MMMM'),
                      (4500, 'MMMMD'),
                      (4888, 'MMMMDCCCLXXXVIII'),
                      (4999, 'MMMCMXCIX') )

```

①

```

class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman dovrebbe fallire con numeri grandi'''
        self.assertRaises(roman8.OutOfRangeError, roman8.to_roman, 5000)

    .
    .
    .

```

②

```

class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman dovrebbe fallire con cifre ripetute troppe volte'''
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, s)

    .
    .
    .

```

③

```

class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n per tutti gli n'''
        for integer in range(1, 5000):

```

④

```
numeral = roman8.to_roman(integer)
result = roman8.from_roman(numeral)
self.assertEqual(integer, result)
```

1. I valori noti già esistenti non cambiano (sono ancora tutti valori ragionevoli da collaudare), ma avete bisogno di aggiungerne alcuni nell'intervallo delle quattro migliaia. Qui ho incluso 4000 (il più corto), 4500 (il secondo più corto), 4888 (il più lungo) e 4999 (il più grande).
2. La definizione di “numero grande” è cambiata. Questo test invocava `to_roman()` con 4000 e si aspettava un errore, ma ora che i valori 4000-4999 sono validi avete bisogno di alzare la soglia a 5000.
3. Anche la definizione di “cifre ripetute troppe volte” è cambiata. Questo test invocava `from_roman()` con 'MMMM' e si aspettava un errore, ma ora che MMMM è considerato un numero romano valido avete bisogno di alzare il valore limite a 'MMMMM'.
4. Il test di consistenza controlla ogni numero nell'intervallo, da 1 fino a 3999. Dato che ora l'intervallo si è allargato, anche questo ciclo `for` deve essere aggiornato per arrivare fino a 4999.

Ora i vostri test sono aggiornati rispetto ai nuovi requisiti, ma il vostro codice non lo è, quindi quando li eseguite aspettatevi diversi fallimenti.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest9.py -v
from_roman dovrebbe fallire con una stringa vuota ... ok
from_roman dovrebbe fallire con antecedenti malformati ... ok
from_roman dovrebbe fallire con ingressi diversi da stringhe ... ok
from_roman dovrebbe fallire con coppie di cifre ripetute ... ok
from_roman dovrebbe fallire con cifre ripetute troppe volte ... ok
from_roman dovrebbe dare un risultato noto con un ingresso noto ... ERROR ①
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ERROR ②
from_roman(to_roman(n))==n per tutti gli n ... ERROR ③
to_roman dovrebbe fallire con numeri negativi ... ok
to_roman dovrebbe fallire con numeri non interi ... ok
to_roman dovrebbe fallire con numeri grandi ... ok
to_roman dovrebbe fallire con il numero 0 ... ok
```

```
=====
```

```
ERROR: from_roman dovrebbe dare un risultato noto con un ingresso noto
```

```
-----
```

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

```
File "romantest9.py", line 82, in test_from_roman_known_values
```

```
    result = roman9.from_roman(numeral)
```

```
File "C:\home\diveintopython3\esempi\roman9.py", line 60, in from_roman
```

```
    raise InvalidRomanNumeralError('Numero romano non valido: {0}'.format(s))
```

```
roman9.InvalidRomanNumeralError: Numero romano non valido: MMMM
```

```
=====
```

```
ERROR: to_roman dovrebbe dare un risultato noto con un ingresso noto
```

```
-----
```

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

```
File "romantest9.py", line 76, in test_to_roman_known_values
```

```
    result = roman9.to_roman(integer)
```

```
File "C:\home\diveintopython3\esempi\roman9.py", line 42, in to_roman
```

```
    raise OutOfRangeError("numero fuori dall'intervallo (deve essere tra 1 e 3999)")
```

```
roman9.OutOfRangeError: numero fuori dall'intervallo (deve essere tra 1 e 3999)
```

```
=====
ERROR: from_roman(to_roman(n))==n per tutti gli n
-----

Traceback (most recent call last):
  File "romantest9.py", line 131, in testSanity
    numeral = roman9.to_roman(integer)
  File "C:\home\diveintopython3\esempi\roman9.py", line 42, in to_roman
    raise OutOfRangeError("numero fuori dall'intervallo (deve essere tra 1 e 3999)")
roman9.OutOfRangeError: numero fuori dall'intervallo (deve essere tra 1 e 3999)

-----

Ran 12 tests in 0.171s

FAILED (errors=3)
```

1. Il test sui valori noti per `from_roman()` fallirà non appena arriva a 'MMMM', perché `from_roman()` pensa ancora che questo sia un numero romano non valido.
2. Il test sui valori noti per `to_roman()` fallirà non appena arriva a 4000, perché `to_roman()` pensa ancora che questo numero sia fuori dall'intervallo di validità.
3. Anche il controllo di consistenza fallirà non appena arriva a 4000, perché `to_roman()` pensa ancora che questo numero sia fuori dall'intervallo di validità.

Ora che i vostri test falliscono a causa dei nuovi requisiti, potete pensare a correggere il codice per riallinearlo con i test. (Quando cominciate a utilizzare i test di unità per la prima volta, potrebbe sembrarvi strano che il codice sotto collaudo non sia mai “avanti” rispetto ai test. Mentre il codice è indietro avete ancora del lavoro da fare, e appena lo avete rimesso in pari con i test potete smettere di programmare. Dopo averci fatto l'abitudine, vi chiederete come facevate a programmare senza i test.)

```

roman_numeral_pattern = re.compile('''
    ^                # inizio della stringa
    M{0,4}           # migliaia - da 0 a 4 M ①
    (CM|CD|D?C{0,3}) # centinaia - 900 (CM), 400 (CD), 0-300 (da 0 a 3 C),
                      # o 500-800 (D, seguita da 0 fino a 3 C)
    (XC|XL|L?X{0,3}) # decine - 90 (XC), 40 (XL), 0-30 (da 0 a 3 X),
                      # o 50-80 (L, seguita da 0 fino a 3 X)
    (IX|IV|V?I{0,3}) # unità - 9 (IX), 4 (IV), 0-3 (da 0 a 3 I),
                      # o 5-8 (V, seguita da 0 fino a 3 I)
    $                # fine della stringa
''', re.VERBOSE)

def to_roman(n):
    '''converte un intero in un numero romano'''
    if not (0 < n < 5000): ②
        raise OutOfRangeError("numero fuori dall'intervallo (deve essere tra 1 e 4999)")
    if not isinstance(n, int):
        raise NotIntegerError('numeri non interi non possono essere convertiti')

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result

def from_roman(s):
    .
    .
    .

```

- I. Non avete bisogno di fare alcun cambiamento alla funzione `from_roman()`. L'unico cambiamento è a `roman_numeral_pattern`. Se guardate da vicino la prima sezione dell'espressione regolare, noterete che ho cambiato il massimo numero di caratteri M opzionali da 3 a 4. Questa modifica renderà legali i numeri romani

equivalenti fino a 4999 anziché 3999. La funzione `from_roman()` è in effetti completamente generica: controlla semplicemente i caratteri ripetuti nei numeri romani e li somma, senza preoccuparsi di quante volte si ripetano. In precedenza la funzione non poteva gestire 'MMMM' solo perché l'avevate esplicitamente fermata utilizzando la corrispondenza con il pattern dell'espressione regolare.

2. La funzione `to_roman()` ha bisogno solo di un piccolo cambiamento nel controllo sull'intervallo. Laddove controllavate che fosse $0 < n < 4000$, ora controllate che sia $0 < n < 5000$. Anche il messaggio di errore che sollevate tramite `raise` va modificato per riflettere il nuovo intervallo accettabile (tra 1 e 4999 anziché tra 1 e 3999). Non avete bisogno di fare alcun cambiamento al resto della funzione, perché gestisce già i nuovi casi. (Aggiunge tranquillamente 'M' per ogni migliaia che trova; ricevuto in ingresso 4000, restituirà in uscita 'MMMM'. In precedenza la funzione non poteva farlo solo perché l'avevate esplicitamente fermata tramite il controllo sull'intervallo.)

Potreste non essere convinti che questi due piccoli cambiamenti siano tutto ciò di cui avete bisogno. Ehi, non siete obbligati a fidarvi della mia parola: controllate voi stessi.

```
you@localhost:~/diveintopython3/esempi$ python3 romantest9.py -v
from_roman dovrebbe fallire con una stringa vuota ... ok
from_roman dovrebbe fallire con antecedenti malformati ... ok
from_roman dovrebbe fallire con ingressi diversi da stringhe ... ok
from_roman dovrebbe fallire con coppie di cifre ripetute ... ok
from_roman dovrebbe fallire con cifre ripetute troppe volte ... ok
from_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
from_roman(to_roman(n))==n per tutti gli n ... ok
to_roman dovrebbe fallire con numeri negativi ... ok
to_roman dovrebbe fallire con numeri non interi ... ok
to_roman dovrebbe fallire con numeri grandi ... ok
to_roman dovrebbe fallire con il numero 0 ... ok
```

```
-----
Ran 12 tests in 0.203s
```

OK ①

I. Tutti i test hanno successo. Potete smettere di programmare.

Un insieme completo di test di unità ci permette di non dover mai fare affidamento su un programmatore che dice: “Fidati di me.”



10.3. REFACTORING

La cosa migliore di un collaudo di unità completo non è la sensazione che provate quando tutti i vostri test hanno finalmente successo, e nemmeno la sensazione che provate quando qualcun altro vi accusa di aver guastato il suo codice e voi potete effettivamente *dimostrare* di non averlo fatto. La cosa migliore dei test di unità è che vi danno la libertà di applicare sistematicamente il refactoring.

Il refactoring è il procedimento tramite il quale si modifica codice già funzionante allo scopo di farlo funzionare meglio. Di solito, “meglio” significa “più velocemente”, ma può anche voler dire “usando meno memoria”, oppure “usando meno spazio su disco”, o semplicemente “in maniera più elegante”. Qualsiasi cosa voglia dire per voi, per il vostro progetto, nel vostro ambiente, il refactoring è importante per la salute a lungo termine di qualsiasi programma.

Nel nostro caso, “meglio” significa sia “più velocemente” che “in maniera più facile da mantenere”. Nello specifico, la funzione `from_roman()` è lenta e più complessa di quanto vorrei, a causa di quella grossa e disgustosa espressione regolare che usate per validare i numeri romani. Ora, potreste pensare: “Certo, l’espressione regolare è lunga e intricata, ma in quale altro modo si suppone che io controlli che una stringa arbitraria sia un numero romano valido?”

Risposta: di numeri romani validi ce ne sono solo 5000; perché non costruite semplicemente una tabella di ricerca? Questa idea diventa ancora migliore nel momento in cui realizzate che *non avete per niente bisogno di usare un’espressione regolare*. Mentre costruite la tabella di ricerca per convertire i numeri interi in numeri romani potete costruire la tabella di ricerca inversa per convertire i numeri romani in interi. Nel momento in cui avete bisogno di controllare se una stringa arbitraria è un numero romano valido avrete già raccolto tutti i numeri romani validi. La “verifica” si riduce a un singolo accesso a un dizionario.

E la cosa migliore di tutte è che avete già un insieme completo di test di unità. Potete cambiare metà del codice nel modulo ma i test di unità rimarranno gli stessi. Questo significa che potete dimostrare — a voi stessi e agli altri — che il nuovo codice funziona tanto bene quanto quello originale.

```

class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
class InvalidRomanNumeralError(ValueError): pass

roman_numeral_map = (('M', 1000),
                      ('CM', 900),
                      ('D', 500),
                      ('CD', 400),
                      ('C', 100),
                      ('XC', 90),
                      ('L', 50),
                      ('XL', 40),
                      ('X', 10),
                      ('IX', 9),
                      ('V', 5),
                      ('IV', 4),
                      ('I', 1))

to_roman_table = [ None ]
from_roman_table = {}

def to_roman(n):
    '''converte un intero in un numero romano'''
    if not (0 < n < 5000):
        raise OutOfRangeError("numero fuori dall'intervallo (deve essere tra 1 e 4999)")
    if int(n) != n:
        raise NotIntegerError('numeri non interi non possono essere convertiti')
    return to_roman_table[n]

def from_roman(s):
    '''converte un numero romano in un intero'''
    if not isinstance(s, str):
        raise InvalidRomanNumeralError("L'ingresso deve essere una stringa")
    if not s:

```

```

        raise InvalidRomanNumeralError('La stringa in ingresso non può essere vuota')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError('Numero romano non valido: {0}'.format(s))
    return from_roman_table[s]

def build_lookup_tables():
    def to_roman(n):
        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer)
        to_roman_table.append(roman_numeral)
        from_roman_table[roman_numeral] = integer

build_lookup_tables()

```

Dividiamo il programma in frammenti più comprensibili. Teoricamente, la riga più importante è l'ultima:

```
build_lookup_tables()
```

Noterete che è una chiamata di funzione, ma non è preceduta da alcuna istruzione `if`. Questo non è un blocco `if __name__ == '__main__':`; la funzione viene invocata *quando il modulo è importato*. (È importante capire che i moduli sono importati solamente una volta e poi vengono salvati in memoria. Se importate un modulo già importato non succede nulla. Quindi questo codice verrà invocato solo la prima volta che importate questo modulo.)

Cosa fa la funzione `build_lookup_tables()`? Sono contento che lo abbiate chiesto.

```
to_roman_table = [ None ]
from_roman_table = {}
.
.
.
def build_lookup_tables():
    def to_roman(n): ①
        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer) ②
        to_roman_table.append(roman_numeral) ③
        from_roman_table[roman_numeral] = integer
```

1. Questa è una tecnica di programmazione ingegnosa... forse troppo ingegnosa. La funzione `to_roman()` è definita sopra; accede alla tabella di ricerca e ne restituisce i valori. Ma la funzione `build_lookup_tables()` ridefinisce la funzione `to_roman()` per fare effettivamente qualcosa di più (come accadeva nell'esempio precedente, prima che aggiungeste una tabella di ricerca). Nell'ambito della funzione `build_lookup_tables()`, una chiamata a `to_roman()` invocherà questa versione ridefinita. Una volta usciti dalla funzione `build_lookup_tables()`, la versione ridefinita scompare — essa è definita solo nell'ambito locale della funzione `build_lookup_tables()`.
2. Questa riga di codice chiamerà la funzione `to_roman()` ridefinita, che calcola effettivamente il numero romano.

3. Una volta che avete ottenuto il risultato (dalla funzione `to_roman()` ridefinita), potete aggiungere l'intero e il suo numero romano equivalente a entrambe le tabelle di ricerca.

Una volta che le tabelle di ricerca sono popolate, il resto del codice è sia semplice che veloce.

```
def to_roman(n):  
    '''converte un intero in un numero romano'''  
    if not (0 < n < 5000):  
        raise OutOfRangeError("numero fuori dall'intervallo (deve essere tra 1 e 4999)")  
    if int(n) != n:  
        raise NotIntegerError('numeri non interi non possono essere convertiti')  
    return to_roman_table[n] ①
```

```
def from_roman(s):  
    '''converte un numero romano in un intero'''  
    if not isinstance(s, str):  
        raise InvalidRomanNumeralError("L'ingresso deve essere una stringa")  
    if not s:  
        raise InvalidRomanNumeralError('La stringa in ingresso non può essere vuota')  
    if s not in from_roman_table:  
        raise InvalidRomanNumeralError('Numero romano non valido: {0}'.format(s))  
    return from_roman_table[s] ②
```

1. Dopo aver fatto gli stessi controlli di prima sugli estremi, la funzione `to_roman()` si limita a trovare il valore appropriato nella tabella di ricerca e a restituirlo.
2. Similmente, la funzione `from_roman()` si riduce a qualche controllo sugli estremi e a una riga di codice. Niente più espressioni regolari. Niente più cicli. Conversione di complessità $O(1)$ da numeri interi in numeri romani e viceversa.

Ma funziona? Certo che sì, sì che funziona. E posso dimostrarlo.

```

you@localhost:~/diveintopython3/esempi$ python3 romantest10.py -v
from_roman dovrebbe fallire con una stringa vuota ... ok
from_roman dovrebbe fallire con antecedenti malformati ... ok
from_roman dovrebbe fallire con ingressi diversi da stringhe ... ok
from_roman dovrebbe fallire con coppie di cifre ripetute ... ok
from_roman dovrebbe fallire con cifre ripetute troppe volte ... ok
from_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
to_roman dovrebbe dare un risultato noto con un ingresso noto ... ok
from_roman(to_roman(n))==n per tutti gli n ... ok
to_roman dovrebbe fallire con numeri negativi ... ok
to_roman dovrebbe fallire con numeri non interi ... ok
to_roman dovrebbe fallire con numeri grandi ... ok
to_roman dovrebbe fallire con il numero 0 ... ok

```

```

-----
Ran 12 tests in 0.031s

```

①

OK

- I. Non che lo abbiate chiesto, ma è anche veloce! Tipo, almeno 10× più veloce. Naturalmente, questo non è un confronto onesto, perché questa versione impiega più tempo a essere importata (quando costruisce le tabelle di ricerca). Ma dato che viene importata una volta sola, il costo di inizializzazione è ammortizzato su tutte le chiamate delle funzioni `to_roman()` e `from_roman()`. E dato che i test effettuano diverse migliaia di chiamate di funzione (il solo test di consistenza ne fa 10.000), questo risparmio si accumula molto velocemente!

Qual è la morale della storia?

- La semplicità è una virtù.
- Specialmente quando sono coinvolte espressioni regolari.
- I test di unità possono darvi la sicurezza necessaria per effettuare il refactoring su larga scala.



10.4. RIEPILOGO

Il collaudo di unità è un potente concetto che, se adeguatamente implementato, può sia ridurre i costi di manutenzione sia aumentare la flessibilità di ogni progetto a lungo termine. È anche importante capire che il collaudo di unità non è una panacea, un Risolutore Magico di Problemi, o la cosiddetta pallottola d'argento. Scrivere buoni test di unità è difficile e mantenerli aggiornati richiede disciplina (specialmente quando i committenti vi urlano dietro per farvi correggere bug critici). I test di unità non sostituiscono altre forme di collaudo, compresi i test funzionali, i test di integrazione e i test di accettazione per l'utente. Ma sono convenienti, e funzionano, e una volta che li avete visti funzionare vi chiederete come avete fatto a lavorare senza fino a quel momento.

Questi pochi capitoli hanno coperto diversi argomenti, molti dei quali non erano nemmeno specifici per Python. Esistono framework per il collaudo di unità in molti linguaggi, e tutti vi richiedono di applicare una serie di pratiche basate sugli stessi concetti elementari:

- progettate test che siano specifici, automatizzati e indipendenti;
- scrivete i test *prima* del codice che devono collaudare;
- scrivete test che utilizzino dati di ingresso accettabili e verifichino i risultati corretti;
- scrivete test che utilizzino dati di ingresso non accettabili e controllino le corrette risposte di fallimento;
- scrivete e aggiornate i test per riflettere nuovi requisiti;
- applicate sistematicamente il refactoring per migliorare prestazioni, scalabilità, leggibilità, manutenibilità, o qualsiasi altra -ilità che vi manca.

CAPITOLO 11. FILE

“Camminare per nove miglia non è uno scherzo, specialmente se piove.”
— Harry Kemelman, *Nove miglia sotto la pioggia*

11.1. IMMERSIONE!

Occasionalmente, il mio portatile Windows aveva 38.493 file prima che installassi una sola applicazione. L'installazione di Python 3 ha aggiunto quasi 3.000 file a quel totale. I file sono il paradigma di memorizzazione principale di tutti i sistemi operativi più importanti; il concetto è talmente radicato che la maggior parte delle persone avrebbe difficoltà a immaginare un'alternativa. Il vostro computer, metaforicamente parlando, annega nei file.

11.2. LEGGERE DAI FILE DI TESTO

Prima di poter leggere da un file, avete bisogno di aprirlo. Aprire un file in Python non potrebbe essere più facile:

```
a_file = open('esempi/chinese.txt', encoding='utf-8')
```

Python possiede una funzione `open()` predefinita che prende un nome di file come argomento. Qui il nome del file è `'esempi/chinese.txt'`. Ci sono cinque cose interessanti da notare a proposito di questo nome di file.

1. Non è solo il nome di un file, ma è la combinazione di un percorso di directory e di un nome di file. Un'ipotetica funzione di apertura di file potrebbe accettare due argomenti — un percorso di directory e un nome di file — ma la funzione `open()` ne accetta solo uno. In Python, ogni volta che avete bisogno di un “nome di file” potete anche includere un percorso di directory intero o parziale.

2. Il percorso di directory usa i caratteri di slash, ma io non vi ho detto quale sistema operativo stavo usando. Windows usa i caratteri di backslash per denotare le sottodirectory, mentre Mac OS X e Linux usano i caratteri di slash. Ma in Python gli slash funzionano sempre, persino su Windows.
3. Il percorso di directory non comincia con uno slash o una lettera di disco, quindi viene chiamato *percorso relativo*. Potreste chiedervi: relativo a cosa? Sii paziente, cavalletta.
4. È una stringa. Tutti i sistemi operativi moderni (persino Windows!) usano Unicode per memorizzare i nomi di file e directory. Python 3 supporta pienamente i nomi di percorso in codifiche diverse da ASCII.
5. Il file non deve necessariamente trovarsi sul vostro disco locale. Potreste aver montato un disco di rete, oppure quel file potrebbe essere una finzione appartenente a un file system interamente virtuale. Se il vostro computer lo considera un file e può accedervi come a un file, Python può aprirlo.

Ma quella invocazione alla funzione `open()` non si è fermata al nome di file. C'è un altro argomento, chiamato `encoding`. Oh, cavoli, questo suona spaventosamente familiare.

11.2.1. LA CODIFICA DI CARATTERE SOLLEVA LA SUA RIPUGNANTE TESTA

I byte sono byte, i caratteri sono un'astrazione. Una stringa è una sequenza di caratteri Unicode. Ma un file su disco non è una sequenza di caratteri Unicode, bensì una sequenza di byte. Quindi, se leggete un “file di testo” dal disco, in che modo Python converte quella sequenza di byte in una sequenza di caratteri? Decodifica i byte seguendo un particolare algoritmo di codifica di carattere e restituisce una sequenza di caratteri Unicode (altrimenti nota come una stringa).

```
# Questo esempio è stato creato su Windows. Altre piattaforme potrebbero
# comportarsi in maniera diversa, per le ragioni illustrate di seguito.
```

```
>>> file = open('esempi/chinese.txt')
```

```
>>> a_string = file.read()
```

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

```
File "<stdin>", line 1, in <module>
```

```
File "C:\Python31\lib\encodings\cp1252.py", line 23, in decode
```

```
    return codecs.charmap_decode(input,self.errors,decoding_table)[0]
```

```
UnicodeDecodeError: 'charmap' codec can't decode byte 0x8f in position 28: character maps to <undefined>
```

```
>>>
```

Che cos'è appena successo? Non avete specificato una codifica di carattere, quindi Python è obbligato a usare la codifica predefinita. Qual è la codifica predefinita? Se osservate attentamente la traccia dello stack, potete vedere che l'esecuzione si blocca nel file `cp1252.py`, indicando che in questo caso Python sta usando CP-1252 come codifica predefinita. (CP-1252 è una codifica comune sui computer che eseguono Microsoft Windows.) L'insieme di caratteri CP-1252 non supporta i caratteri contenuti in questo file, quindi la lettura fallisce con un brutto errore di tipo `UnicodeDecodeError`.

*La codifica
predefinita
dipende dalla
piattaforma.*

Ma aspettate, le cose sono peggio di così! La codifica predefinita *dipende dalla piattaforma*, quindi questo codice *potrebbe* funzionare sul vostro computer (se la vostra codifica predefinita è UTF-8), ma poi fallirebbe quando lo distribuite a qualcun altro (la cui codifica predefinita è differente, come CP-1252).

☞ Se avete bisogno di ottenere la codifica di carattere predefinita, importate il modulo `locale` e invocate `locale.getpreferredencoding()`. Sul mio portatile Windows la funzione restituisce `'cp1252'`, ma sulla macchina Linux che ho di sopra restituisce `'UTF-8'`. Non riesco a mantenere la consistenza nemmeno in casa mia! I vostri risultati potrebbero essere differenti (persino su Windows) a seconda di quale versione del vostro sistema operativo avete installato e di come sono configurate le impostazioni regionali e di lingua. Questo è il motivo per cui è così importante specificare la codifica ogni volta che aprite un file.

11.2.2. OGGETTI STREAM


Finora tutto quello che sappiamo è che Python ha una funzione predefinita chiamata `open()`. La funzione `open()` restituisce un *oggetto stream*, che è dotato di metodi e attributi per ottenere informazioni su un flusso di caratteri e manipolarlo.

```

>>> a_file = open('esempi/chinese.txt', encoding='utf-8')
>>> a_file.name ①
'esempi/chinese.txt'
>>> a_file.encoding ②
'utf-8'
>>> a_file.mode ③
'r'

```

1. L'attributo `name` riflette il nome che avete passato alla funzione `open()` quando avete aperto il file. Questo nome non viene normalizzato sotto forma di nome di percorso assoluto.
2. Similmente, l'attributo `encoding` riflette la codifica che avete passato alla funzione `open()`. Se non avete specificato la codifica quando avete aperto il file (cattivi sviluppatori!) allora l'attributo `encoding` rifletterà il valore restituito da `locale.getpreferredencoding()`.
3. L'attributo `mode` vi dice in quale modalità è stato aperto il file. Potete passare un parametro `mode` opzionale alla funzione `open()`. Non avete specificato una modalità quando avete aperto questo file, così Python usa `'r'` per default, che significa “apri in sola lettura, in modalità testo”. Come vedrete più avanti in questo capitolo, la modalità del file serve a vari scopi: diverse modalità vi permettono di scrivere su un file, aggiungere dati in fondo al file, o aprire un file in modalità binaria (nel qual caso avrete a che fare con byte invece di stringhe).

 La documentazione per la funzione `open()` elenca tutte le possibili modalità dei file.

11.2.3. LEGGERE DATI DA UN FILE DI TESTO

Dopo aver aperto un file in lettura, a un certo punto vorrete probabilmente leggerne i dati.

```

>>> a_file = open('esempi/chinese.txt', encoding='utf-8')
>>> a_file.read() ①
'Immersione in Python 是为有经验的程序员编写的一本 Python 书。 \n'
>>> a_file.read() ②
''

```

1. Una volta che avete aperto un file (con la codifica corretta), leggere da quel file è semplicemente una questione di invocare il metodo `read()` dell'oggetto stream. Il risultato è una stringa.
2. In modo forse abbastanza sorprendente, leggere ancora dal file non solleva un'eccezione. Python non considera una lettura dopo la fine del file come un errore, ma restituisce semplicemente una stringa vuota.

E se voleste rileggere un file?

```
# continua dall'esempio precedente
>>> a_file.read() ①
''
>>> a_file.seek(0) ②
0
>>> a_file.read(20) ③
'Immersione in Python'
>>> a_file.read(1) ④
' '
>>> a_file.read(1)
'是'
>>> a_file.tell() ⑤
24
```

*Usate sempre
il parametro
encoding
quando aprite
un file.*

1. Dato che vi trovate ancora alla fine del file, ulteriori invocazioni del metodo `read()` dell'oggetto stream restituiscono semplicemente una stringa vuota.
2. Il metodo `seek()` vi sposta sul byte in una specifica posizione di un file.
3. Il metodo `read()` può accettare come parametro opzionale il numero di caratteri da leggere.
4. Se volete, potete anche leggere un carattere alla volta.
5. `20 + 1 + 1 = ... 24?`

Riproviamo ancora.

```
# continua dall'esempio precedente

>>> a_file.seek(21) ①
21
>>> a_file.read(1) ②
'是'
>>> a_file.tell() ③
24
```

1. Spostatevi al 21° byte.
2. Leggete un carattere.
3. Ora vi trovate al 24° byte.

Ve ne siete già accorti? I metodi `seek()` e `tell()` contano sempre i *byte*, ma dato che avete aperto questo file come testo, il metodo `read()` conta i *caratteri*. I caratteri cinesi richiedono più byte per venire codificati in UTF-8. I caratteri inglesi in un file richiedono solo un byte ognuno, quindi potreste essere erroneamente indotti a credere che i metodi `seek()` e `read()` stiano contando le stesse cose. Ma questo è vero solo per alcuni caratteri.

Ma aspettate, le cose peggiorano!

```
>>> a_file.seek(22) ①
22
>>> a_file.read(1) ②

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    a_file.read(1)
  File "C:\Python31\lib\codecs.py", line 300, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode byte 0x98 in position 0: unexpected code byte
```

1. Spostatevi al 22° byte e provate a leggere un carattere.
2. Perché questo fallisce? Perché non c'è alcun carattere al 22° byte. Il carattere più vicino comincia al 21° byte (e prosegue per tre byte). Il tentativo di leggere un carattere nel mezzo fallirà con un errore di tipo `UnicodeDecodeError`.

11.2.4. CHIUDERE I FILE

I file aperti consumano risorse di sistema, e a seconda della modalità del file altri programmi potrebbero non essere in grado di accedervi. È importante chiudere i file non appena avete finito di lavorarci.

```
# continua dall'esempio precedente
>>> a_file.close()
```

Be', *questo* è stato un anticlimax.

L'oggetto stream `a_file` esiste ancora, perché l'invocazione del suo metodo `close()` non distrugge l'oggetto vero e proprio. Ma non è particolarmente utile.

```
# continua dall'esempio precedente
>>> a_file.read() ①
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    a_file.read()
ValueError: I/O operation on closed file.
>>> a_file.seek(0) ②
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    a_file.seek(0)
ValueError: I/O operation on closed file.
>>> a_file.tell() ③
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    a_file.tell()
ValueError: I/O operation on closed file.
>>> a_file.close() ④
>>> a_file.closed ⑤
True
```

I. Non potete leggere da un file chiuso: questa operazione solleva un'eccezione di tipo `IOError`.

2. Non potete neanche effettuare un'operazione di posizionamento su un file chiuso.
3. Non c'è nessuna posizione corrente in un file chiuso, quindi anche il metodo `tell()` fallisce.
4. Forse in modo sorprendente, invocare il metodo `close()` su un oggetto stream il cui file è stato chiuso *non* solleva un'eccezione. Semplicemente, non fa nulla.
5. Gli oggetti stream chiusi hanno almeno un attributo utile: l'attributo `closed` vi confermerà che il file è chiuso.

11.2.5. CHIUDERE I FILE AUTOMATICAMENTE

Gli oggetti stream hanno un metodo `close()` esplicito, ma cosa succede se il vostro codice ha un bug e si blocca prima che possiate chiamare `close()`? Quel file potrebbe teoricamente rimanere aperto per un tempo molto più lungo del necessario. Mentre state effettuando il debug sul vostro computer locale questo non è un grande problema, ma su un server di produzione forse lo è.

Python 2 aveva una soluzione per questo: il blocco `try..finally`. Questo funziona ancora in Python 3, e potreste vederlo nel codice di altre persone o in codice più vecchio che è stato convertito verso Python 3. Ma Python 2.6 ha introdotto una soluzione più pulita, che ora è la soluzione preferita in Python 3: l'istruzione `with`.


`try..finally`
va bene.
`with` *va*
molto meglio.

```
with open('esempi/chinese.txt', encoding='utf-8') as a_file:
    a_file.seek(21)
    a_character = a_file.read(1)
    print(a_character)
```

Questo codice invoca `open()` ma non invoca mai `a_file.close()`. L'istruzione `with` comincia un blocco di codice, come un'istruzione `if` o un ciclo `for`. All'interno di questo blocco di codice, potete usare la variabile `a_file` come l'oggetto stream restituito dalla chiamata a `open()`. Tutti i normali metodi degli oggetti stream

sono disponibili — `seek()`, `read()`, qualsiasi cosa di cui abbiate bisogno. Quando il blocco `with` finisce, *Python* invoca `a_file.close()` automaticamente.

Ecco la sorpresa: a prescindere da come e quando uscite dal blocco `with`, *Python* chiuderà quel file... anche se “uscite” attraverso un’eccezione non gestita. Esatto, anche se il vostro codice solleva un’eccezione e il vostro programma si blocca, quel file verrà chiuso. Garantito.


 In termini tecnici, l’istruzione `with` crea un contesto di esecuzione. In questi esempi, l’oggetto `stream` agisce come un gestore di contesto. *Python* crea l’oggetto `stream` `a_file` e gli dice che sta entrando nel contesto di esecuzione. Quando il blocco di codice dell’istruzione `with` viene completato, *Python* dice all’oggetto `stream` che sta uscendo dal contesto di esecuzione e l’oggetto `stream` invoca il proprio metodo `close()`. Leggete la sezione Classi che possono essere usate in un blocco `with` nell’appendice B per i dettagli.

Non c’è nulla di specifico per i file nell’istruzione `with`: è semplicemente un framework generico per creare contesti di esecuzione e informare gli oggetti sul fatto che stanno entrando o uscendo da un contesto di esecuzione. Se l’oggetto in questione è un flusso, allora esegue utili operazioni relative ai file (come chiudere il file automaticamente). Ma questo comportamento è definito nell’oggetto `stream`, non nell’istruzione `with`. Ci sono molti altri modi per usare i gestori di contesto che non hanno nulla a che fare con i file. Potete persino creare il vostro gestore, come vedrete più avanti in questo capitolo.

11.2.6. LEGGERE I DATI UNA RIGA ALLA VOLTA

Una “riga” di un file di testo è esattamente quello che pensate che sia — digitate alcune parole, premete INVIO e ora vi trovate su una nuova riga. Una riga di testo è una sequenza di caratteri delimitata da... cosa, esattamente? Be’, è complicato, perché i file di testo possono usare diversi caratteri per contrassegnare la fine di una riga. Ogni sistema operativo ha le proprie convenzioni. Alcuni usano un carattere di ritorno a capo, altri usano un carattere di fine riga e alcuni usano entrambi i caratteri al termine di ogni riga.

Ora tirate un sospiro di sollievo, perché *Python gestisce i caratteri di fine riga automaticamente* per default. Se dite “voglio leggere questo file di testo una riga alla volta”, Python scoprirà che tipo di carattere di fine riga viene usato dal file di testo e ogni cosa funzionerà a dovere.

 Se avete bisogno di un controllo più fine su quali caratteri vengono considerati per indicare la fine di una riga, potete passare il parametro opzionale `newline` alla funzione `open()`. Leggete la [documentazione della funzione open\(\)](#) per conoscere tutti i dettagli.

Quindi, come fate effettivamente a farlo? Leggere un file una riga alla volta, intendo. È talmente semplice da essere bello.

```
line_number = 0
with open('esempi/favorite-people.txt', encoding='utf-8') as a_file: ①
    for a_line in a_file: ②
        line_number += 1
        print('{:>4} {}'.format(line_number, a_line.rstrip())) ③
```

1. Usando il pattern with, aprite il file in sicurezza e lasciate che Python lo chiuda per voi.
2. Per leggere un file una riga alla volta usate un ciclo `for`. Questo è tutto. Oltre a possedere metodi espliciti come `read()`, *l'oggetto stream è anche un iteratore* che restituisce una singola riga ogni volta che gli chiedete un valore.
3. Usando il metodo `format()` delle stringhe, potete stampare il numero di riga e la riga stessa. La specifica di formato `{:>4}` significa “stampa l'argomento giustificato a destra all'interno di 4 spazi”. La variabile `a_line` contiene la riga completa, ritorni a capo e tutto quanto. Il metodo `rstrip()` delle stringhe rimuove gli spazi bianchi in coda, compresi i caratteri di ritorno a capo.

```
you@localhost:~/diveintopython3$ python3 esempi/online.py
```

```
1 Dora
2 Ethan
3 Wesley
4 John
5 Anne
6 Mike
7 Chris
8 Sarah
9 Alex
10 Lizzie
```

Avete ottenuto questo errore?

```
you@localhost:~/diveintopython3$ python3 esempi/online.py
```

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

```
File "esempi/online.py", line 4, in <module>
```

```
    print('{:>4} {}'.format(line_number, a_line.rstrip()))
```

```
ValueError: zero length field name in format
```

Se è così, probabilmente state usando Python 3.0. Dovreste davvero aggiornarvi a Python 3.1.

Python 3.0 supportava la formattazione di stringhe, ma solo con specifiche di formato esplicitamente numerate. Python 3.1 vi permette di omettere gli indici degli argomenti nelle vostre specifiche di formato. Per fare un confronto, ecco la versione compatibile con Python 3.0:

```
print('{0:>4} {1}'.format(line_number, a_line.rstrip()))
```



11.3. SCRIVERE SUI FILE DI TESTO

Potete scrivere sui file in modo simile a come li leggete.

Prima aprite un file e ottenete un oggetto stream, poi usate i metodi dell'oggetto stream per scrivere dati sul file, infine chiudete il file.

Per aprire un file in modo da scriverci sopra, usate la funzione `open()` e specificate la modalità di scrittura. Ci sono due modalità per scrivere su un file.

- La modalità di “scrittura” sovrascriverà il file. Passate `mode='w'` alla funzione `open()`.
- La modalità di “aggiunta” aggiungerà i dati alla fine del file. Passate `mode='a'` alla funzione `open()`.

In entrambe le modalità, il file verrà automaticamente creato se non esiste già, quindi non c'è mai bisogno di alcuna funzione che, nel caso il file non esista, tergiversi creando un nuovo file vuoto giusto in modo che voi possiate aprirlo per la prima volta. Dovete solo aprire un file e cominciare a scrivere.

Dovreste sempre chiudere un file non appena avete finito di scriverlo, in modo da rilasciare il puntatore al file e assicurarvi che i dati vengano effettivamente scritti sul disco. Come quando leggete i dati da un file, potete invocare il metodo `close()` dell'oggetto stream, oppure potete usare l'istruzione `with` e lasciare che Python chiuda il file per voi. Scommetto che riuscite a indovinare quale tecnica vi consiglio di usare.

*Dovete solo
aprire un file
e cominciare
a scrivere.*

```

>>> with open('test.log', mode='w', encoding='utf-8') as a_file: ①
...     a_file.write('il test ha funzionato') ②
>>> with open('test.log', encoding='utf-8') as a_file:
...     print(a_file.read())
il test ha funzionato

>>> with open('test.log', mode='a', encoding='utf-8') as a_file: ③
...     a_file.write('ancora una volta')
>>> with open('test.log', encoding='utf-8') as a_file:
...     print(a_file.read())
il test ha funzionatoancora una volta ④

```

1. Cominciate baldanzosamente creando il nuovo file `test.log` (oppure sovrascrivendo il file esistente) e aprendo il file per la scrittura. Usare il parametro `mode='w'` significa aprire il file in scrittura. Sì, questo è tanto pericoloso quanto sembra. Spero che non vi interessassero i contenuti precedenti di quel file (se c'erano) perché quei dati ora sono scomparsi.
2. Potete aggiungere dati al file appena aperto con il metodo `write()` dell'oggetto stream restituito dalla funzione `open()`. Dopo che il blocco `with` si è concluso, Python chiude automaticamente il file.
3. È stato così divertente, facciamolo ancora. Ma questa volta con `mode='a'`, per aggiungere i dati in coda al file invece di sovrascriverlo. Questa modalità non danneggerà *mai* i contenuti esistenti del file.
4. Il file `test.log` ora contiene sia la riga originale che avete scritto sia la seconda riga che avete aggiunto. Notate anche che i caratteri di ritorno a capo e di fine riga non sono inclusi. Dato che entrambe le volte non li avete scritti esplicitamente sul file, il file non li contiene. Potete scrivere un carattere di ritorno a capo usando `'\r'` e/o un carattere di fine riga usando `'\n'`. Dato che non avete fatto né l'una né l'altra cosa, tutto quello che avete scritto sul file è finito su un'unica riga.

11.3.1. LA CODIFICA DI CARATTERE, ANCORA UNA VOLTA

Avete notato il parametro `encoding` che è stato passato alla funzione `open()` mentre stavate aprendo il file in scrittura? È importante, non dimenticatevelo mai! Come avete visto all'inizio di questo capitolo, i file non contengono *stringhe*, ma contengono *byte*. Leggere una “stringa” da un file di testo funziona solo perché avete detto a Python quale codifica usare per leggere un flusso di byte e convertirlo in una stringa. Scrivere testo su un file presenta il problema inverso. Non potete scrivere caratteri su un file: i caratteri sono un'astrazione. Per essere in grado di scrivere sul file, Python ha bisogno di convertire la vostra stringa in una

sequenza di byte. L'unico modo per essere sicuri di effettuare la conversione corretta è quello di specificare il parametro encoding quando aprite il file in scrittura.



11.4. FILE BINARI

Non tutti i file contengono testo. Alcuni contengono immagini del mio cane.



```
>>> an_image = open('esempi/beauregard.jpg', mode='rb') ①
>>> an_image.mode ②
'rb'
>>> an_image.name ③
'esempi/beauregard.jpg'
>>> an_image.encoding ④

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: '_io.BufferedReader' object has no attribute 'encoding'
```

1. Aprire un file in modalità binaria è semplice ma sottile. L'unica differenza rispetto ad aprirlo in modalità testo è che il parametro mode contiene un carattere 'b'.
2. L'oggetto stream che ottenete dall'apertura del file in modalità binaria è dotato di molti degli stessi attributi, incluso l'attributo mode che riflette il parametro mode passato alla funzione open().
3. Gli oggetti stream binari hanno anche un attributo name, esattamente come gli oggetti stream testuali.
4. Ecco una differenza, però: un oggetto stream binario non ha l'attributo encoding. Questo ha senso, giusto? State leggendo (o scrivendo) byte, non stringhe, quindi Python non ha nessuna conversione da fare. Quello che ottenete da un file binario è esattamente quello che ci mettete dentro, e nessuna conversione è necessaria.

Vi ho detto che state leggendo byte? Oh sì, è proprio così.

```
# continua dall'esempio precedente
```

```
>>> an_image.tell()
```

```
0
```

```
>>> data = an_image.read(3) ①
```

```
>>> data
```

```
b'\xff\xd8\xff'
```

```
>>> type(data) ②
```

```
<class 'bytes'>
```

```
>>> an_image.tell() ③
```

```
3
```

```
>>> an_image.seek(0)
```

```
0
```

```
>>> data = an_image.read()
```

```
>>> len(data)
```

```
3150
```

1. Come con i file di testo, potete leggere i file binari un pezzo alla volta. Ma c'è una differenza cruciale...
2. ...state leggendo byte, non stringhe. Dato che avete aperto il file in modalità binaria, il metodo `read()` accetta *il numero di byte da leggere*, non il numero di caratteri.
3. Questo significa che non c'è mai uno scarto inatteso tra il numero che avete passato al metodo `read()` e l'indice di posizione che ottenete dal metodo `tell()`. Il metodo `read()` legge byte e i metodi `seek()` e `tell()` tengono traccia del numero dei byte letti. Per quanto riguarda i file binari, si trovano sempre in accordo.



11.5. OGGETTI STREAM PER SORGENTI DIVERSE DAI FILE

Immaginate di scrivere una libreria e che una delle vostre funzioni di libreria debba leggere alcuni dati da un file. La funzione potrebbe semplicemente prendere il nome di file sotto forma di stringa, aprire il file in lettura, leggerlo e chiuderlo prima di uscire. Ma non dovrete fare in questo modo. Invece, la vostra API dovrebbe accettare *un oggetto stream arbitrario*.

Nel caso più semplice, un oggetto stream è qualsiasi cosa con un metodo `read()` che accetta un parametro opzionale `size` e restituisce una stringa. Quando viene chiamato senza parametro `size`, il metodo `read()` dovrebbe leggere tutto quello che c'è da leggere dalla sorgente in ingresso e restituire tutti i dati come un singolo valore. Quando viene chiamato con il parametro `size`, il metodo legge tanti dati quanti sono stati indicati e restituisce quei dati. Quando viene nuovamente chiamato, riprende da dove era rimasto e restituisce il blocco di dati successivo.

Questo sembra esattamente l'oggetto stream che ottenete aprendo un file reale. La differenza è che *non vi limitate ai file reali*. La sorgente in ingresso che viene “letta” potrebbe essere qualsiasi cosa: una pagina web, una stringa in memoria, persino l'uscita di un altro programma. Fino a quando le vostre funzioni accettano un oggetto stream e invocano semplicemente il metodo `read()` dell'oggetto, potete operare su qualsiasi sorgente in ingresso che si comporta come un file, senza usare codice specifico per gestire ogni singolo tipo di ingresso.

*Per leggere
da un falso
file vi basta
chiamare
`read()`.*

```

>>> a_string = 'PapayaWhip è il nuovo nero.'

>>> import io                                ①

>>> a_file = io.StringIO(a_string)           ②

>>> a_file.read()                             ③
'PapayaWhip è il nuovo nero.'

>>> a_file.read()                             ④
''

>>> a_file.seek(0)                           ⑤
0

>>> a_file.read(10)                          ⑥
'PapayaWhip'


>>> a_file.tell()
10

>>> a_file.seek(16)
16

>>> a_file.read()
'nuovo nero.'

```

1. Il modulo `io` definisce la classe `StringIO`, che potete usare per trattare una stringa in memoria come un file.
2. Per creare un oggetto stream a partire da una stringa, create un'istanza della classe `io.StringIO()` e passatele la stringa che volete usare come dati del vostro “file”. Ora avete un oggetto stream e potete eseguire qualsiasi operazione relativa ai flussi su di esso.
3. Invocare il metodo `read()` “legge” l'intero “file” e nel caso di un oggetto `StringIO` restituisce semplicemente la stringa originale.
4. Esattamente come con un vero file, invocare ancora il metodo `read()` restituisce una stringa vuota.
5. Potete esplicitamente tornare all'inizio della stringa con un'operazione di posizionamento, esattamente come accade per un vero file, usando il metodo `seek()` dell'oggetto `StringIO`.
6. Potete anche leggere la stringa a blocchi, passando un parametro `size` al metodo `read()`.

 `io.StringIO` vi permette di trattare una stringa come un file di testo. Esiste anche una classe `io.BytesIO` che vi permette di trattare un array di byte come un file binario.

11.5.1. LAVORARE CON I FILE COMPRESSI

La libreria standard di Python contiene moduli che supportano le operazioni di lettura e scrittura sui file compressi. Esistono un certo numero di schemi di compressione differenti; i due più popolari su sistemi diversi da Windows sono gzip e bzip2. (Potreste anche avere incontrato archivi PKZIP e archivi GNU Tar. Python possiede moduli anche per quelli.)

Il modulo `gzip` vi permette di creare un oggetto stream per leggere o scrivere un file compresso con `gzip`. L'oggetto stream che vi fornisce supporta il metodo `read()` (se lo avete aperto in lettura) o il metodo `write()` (se lo avete aperto in scrittura). Questo significa che potete usare i metodi che avete già imparato per i normali file allo scopo di *leggere o scrivere direttamente un file compresso con gzip* senza creare un file temporaneo dove memorizzare i dati decompressi.

Come bonus aggiuntivo, il modulo supporta anche l'istruzione `with`, quindi potete lasciare che sia Python a chiudere automaticamente il vostro file `gzip` quando avete finito di lavorare.

```
you@localhost:~$ python3
```

```
>>> import gzip
```

```
>>> with gzip.open('out.log.gz', mode='wb') as z_file: ①
```

```
...     z_file.write('Camminare per nove miglia non è uno scherzo, specialmente se piove.'.encode('utf-8'))
```

```
...
```

```
>>> exit()
```

```
you@localhost:~$ ls -l out.log.gz ②
```

```
-rw-r--r--  1 you  you    93 2009-07-19 14:29 out.log.gz
```

```
you@localhost:~$ gunzip out.log.gz ③
```

```
you@localhost:~$ cat out.log ④
```

```
Camminare per nove miglia non è uno scherzo, specialmente se piove.
```

1. Dovreste sempre aprire i file compressi con `gzip` in modalità binaria. (Notate il carattere `'b'` nell'argomento `mode`.)
2. Ho realizzato questo esempio su Linux. Se non avete familiarità con la riga di comando, sappiate che questo comando mostra “l'elenco esteso” del file `gzip` che avete appena creato nella Shell Python. Questo elenco

mostra che il file esiste (bene) e che è di 93 byte. Questo file è in realtà più grande della stringa con cui avete cominciato! Il formato gzip include un'intestazione di lunghezza fissa che contiene alcuni metadati sul file, quindi è inefficiente per file estremamente piccoli.

3. Il comando `gunzip` (pronunciato “gee-unzip” in inglese) decompime il file e memorizza i contenuti in un nuovo file chiamato con lo stesso nome del file compresso ma senza l'estensione `.gz`.
4. Il comando `cat` mostra i contenuti di un file. Questo file contiene la stringa che avevate originariamente scritto direttamente sul file compresso `out.log.gz` dall'interno della Shell Python.

Avete ottenuto questo errore?

```
>>> with gzip.open('out.log.gz', mode='wb') as z_file:
...     z_file.write('Camminare per nove miglia non è [...] se piove.'.encode('utf-8'))
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'GzipFile' object has no attribute '__exit__'
```

Se è così, probabilmente state usando Python 3.0. Dovreste davvero aggiornarvi a Python 3.1.

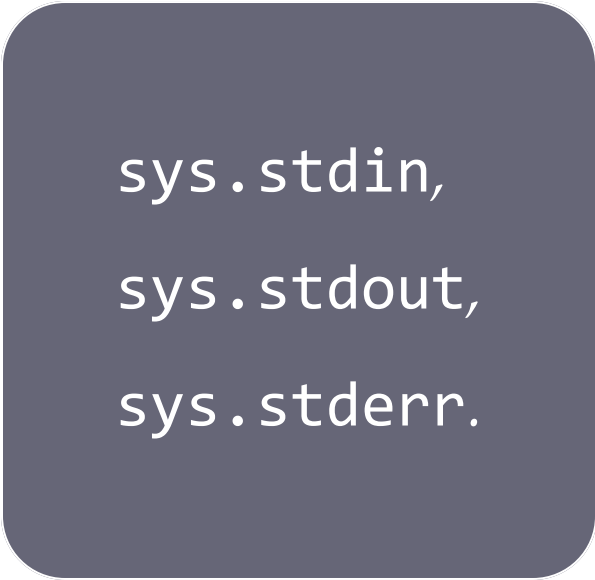
Il modulo `gzip` incluso in Python 3.0 non supportava l'uso di un file `gzip` come gestore di contesto. Python 3.1 ha aggiunto la possibilità di usare oggetti file `gzip` in un'istruzione `with`.



11.6. I CANALI STANDARD DI INGRESSO, USCITA ED ERRORE

I guru della riga di comando conoscono già il concetto di canale standard di ingresso, uscita ed errore. Questa sezione è per tutti gli altri.

I canali di uscita e di errore standard (comunemente abbreviati come `stdout` e `stderr`) sono canali predefiniti in ogni sistema di tipo UNIX, compresi Mac OS X e Linux. Quando invocate la funzione `print()`, quello che state stampando viene inviato al canale `stdout`. Quando il vostro programma si blocca e stampa una traccia dello stack di esecuzione, ciò che viene stampato è inviato al canale `stderr`. Per default, entrambi i canali sono collegati alla finestra di terminale in cui state lavorando, così quando il vostro programma stampa qualcosa vedrete il messaggio nella vostra finestra di terminale, e quando un programma si blocca vedrete anche la traccia dello stack di esecuzione nella stessa finestra. Nella Shell Python grafica, i canali `stdout` e `stderr` corrispondono per default alla vostra “finestra interattiva”.



```
sys.stdin,  
sys.stdout,  
sys.stderr.
```

```
>>> for i in range(3):  
...     print('PapayaWhip')           ①  
PapayaWhip  
PapayaWhip  
PapayaWhip  
  
>>> import sys  
>>> for i in range(3):  
...     sys.stdout.write('è il')      ②  
è ilè ilè il  
  
>>> for i in range(3):  
...     sys.stderr.write('nuovo nero') ③  
nuovo neronuovo neronuovo nero
```

I. La funzione `print()`, in un ciclo. Niente di sorprendente qui.

2. `stdout` è definito nel modulo `sys` ed è un oggetto stream. Invocare la sua funzione `write()` stamperà qualunque stringa le passiate. In effetti, questo è ciò che la funzione `print()` fa in realtà: aggiunge un ritorno a capo alla fine della stringa che state stampando e invoca `sys.stdout.write()`.
3. Nel caso più semplice, `sys.stdout` e `sys.stderr` inviano i loro dati nello stesso posto: un IDE Python (se ne state usando uno) oppure il terminale (se state eseguendo Python dalla riga di comando). Come il canale di uscita standard, il canale di errore standard non aggiunge i ritorni a capo per voi, perciò se li volete dovrete scrivere i caratteri di ritorno a capo.

`sys.stdout` e `sys.stderr` sono oggetti stream, ma sono a sola scrittura. Il tentativo di invocare il loro metodo `read()` solleverà sempre un'eccezione di tipo `IOError`.

```
>>> import sys
>>> sys.stdout.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: not readable
```

11.6.1. REDIRIGERE IL CANALE STANDARD DI USCITA

`sys.stdout` e `sys.stderr` sono oggetti stream, sebbene supportino solo le operazioni di scrittura. Ma non sono costanti, bensì variabili. Questo significa che potete assegnare loro un nuovo valore — qualsiasi altro oggetto stream — per redirigere i loro messaggi.

```

import sys

class RedirectStdoutTo:
    def __init__(self, out_new):
        self.out_new = out_new

    def __enter__(self):
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):
        sys.stdout = self.out_old

print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')

```

Controllate:

```

you@localhost:~/diveintopython3/esempi$ python3 stdout.py
A
C
you@localhost:~/diveintopython3/esempi$ cat out.log
B

```

Avete ottenuto questo errore?

```

you@localhost:~/diveintopython3/esempi$ python3 stdout.py
File "stdout.py", line 15
    with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
                                                                    ^
SyntaxError: invalid syntax

```

Se è così, probabilmente state usando Python 3.0. Dovreste davvero aggiornarvi a Python 3.1.

Python 3.0 supportava l'istruzione `with`, ma ogni istruzione poteva usare solo un gestore di contesto. Python 3.1 vi permette di concatenare molteplici gestori di contesto in una singola istruzione `with`.

Esaminiamo l'ultima parte per prima.

```
print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

Questa è un'istruzione `with` complicata. Permettetemi di riscriverla in qualcosa di più riconoscibile.

```
with open('out.log', mode='w', encoding='utf-8') as a_file:
    with RedirectStdoutTo(a_file):
        print('B')
```

Come mostrato dalla riscrittura, avete effettivamente *due* istruzioni `with`, una annidata nell'ambito dell'altra. L'istruzione `with` “esterna” dovrebbe esservi familiare ormai: apre in scrittura un file di testo codificato in UTF-8 chiamato `out.log` e assegna l'oggetto `stream` a una variabile chiamata `a_file`. Ma questa non è l'unica cosa strana qui.

```
with RedirectStdoutTo(a_file):
```

Dov'è la clausola `as`? L'istruzione `with` in effetti non ne richiede una. Esattamente come potete invocare una funzione e ignorare il suo valore di ritorno, potete avere un'istruzione `with` che non assegna il contesto di `with` a una variabile. In questo caso, siete solo interessati agli effetti collaterali del contesto `RedirectStdoutTo`.

Quali sono questi effetti collaterali? Diamo un'occhiata all'interno della classe `RedirectStdoutTo`. Questa classe è un gestore di contesto personalizzato. Qualsiasi classe può diventare un gestore di contesto definendo due metodi speciali: `__enter__()` ed `__exit__()`.

```
class RedirectStdoutTo:

    def __init__(self, out_new):    ①
        self.out_new = out_new

    def __enter__(self):            ②
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):      ③
        sys.stdout = self.out_old
```

1. Il metodo `__init__()` viene chiamato immediatamente dopo che un'istanza è stata creata. Accetta come parametro l'oggetto stream che volete usare come canale standard di uscita per la vita del contesto. Questo metodo non fa altro che salvare l'oggetto stream in una variabile di istanza in modo che altri metodi possano usarlo più tardi.
2. Il metodo `__enter__()` è un metodo speciale per le classi; l'interprete Python lo invoca quando entra in un contesto (cioè all'inizio dell'istruzione `with`). Questo metodo salva il valore corrente di `sys.stdout` in `self.out_old`, poi reindirizza il canale standard di uscita assegnando `self.out_new` a `sys.stdout`.
3. Il metodo `__exit__()` è un altro metodo speciale per le classi; l'interprete Python lo invoca quando esce da un contesto (cioè alla fine dell'istruzione `with`). Questo metodo ripristina il canale standard di uscita al suo valore originale assegnando il valore salvato in `self.out_old` a `sys.stdout`.

Riepilogando:

```
print('A')    ①
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file): ②
    print('B')    ③
print('C')    ④
```

1. Questo stamperà nella “finestra interattiva” del vostro IDE (o sul terminale, se avete invocato lo script dalla riga di comando).
2. Questa istruzione with prende *una lista di contesti separati da virgole* che agisce come una serie di blocchi with annidati. Il primo contesto nella lista è il blocco “più esterno”, l’ultimo è il blocco “più interno”. Il primo contesto apre un file, il secondo contesto dirige `sys.stdout` all’oggetto stream che è stato creato nel primo contesto.
3. Dato che questa funzione `print()` viene invocata nei contesti creati dall’istruzione with, non stamperà sullo schermo ma scriverà sul file `out.log`.
4. Il blocco di codice with è finito. Python ha detto a tutti i gestori di contesto di fare qualunque cosa facciano al momento di uscire da un contesto. I gestori di contesto formano una pila. Uscendo, il secondo contesto ha ripristinato il valore originale di `sys.stdout`, poi il primo contesto ha chiuso il file chiamato `out.log`. Dato che il canale standard di uscita è stato ripristinato al suo valore originale, l’invocazione della funzione `print()` stamperà ancora una volta sullo schermo.

La redirectione del canale standard di errore funziona esattamente allo stesso modo, usando `sys.stderr` anziché `sys.stdout`.



11.7. LETTURE DI APPROFONDIMENTO

- Leggere e scrivere i file nel tutorial su Python.org
- Il modulo io
- Gli oggetti stream
- I tipi di gestori di contesto
- `sys.stdout` e `sys.stderr`
- FUSE su Wikipedia

CAPITOLO 12. XML

“ Sotto l’arcontato di Aristecmo, Dracone stabilì le sue leggi. ”

— Aristotele

12.1. IMMERSIONE!

Per la maggior parte, i capitoli di questo libro sono stati costruiti attorno al codice di un programma di esempio. Ma XML non ha a che fare con il codice; ha a che fare con i dati. Uno degli usi più comuni di XML è il “syndication feed” che elenca gli ultimi articoli di un blog, un forum, o un altro sito web frequentemente aggiornato. La maggior parte dei software più popolari per la gestione di contenuti web è in grado di produrre un feed e aggiornarlo ogni volta che vengono pubblicati nuovi messaggi, discussioni, o articoli. Potete seguire un singolo blog “abbonandovi” al suo feed, e potete seguire più di un blog alla volta utilizzando un “aggregatore di feed” dedicato come Google Reader.

Qui di seguito, dunque, trovate il documento XML che contiene i dati con cui lavoreremo in questo capitolo. È un feed — nello specifico, un syndication feed in formato Atom.

```

<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'>
  <title>dive into mark</title>
  <subtitle>attualmente tra una dipendenza e l'altra</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
  <link rel='self' type='application/atom+xml' href='http://diveintomark.org/feed/'/>
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Immersione nella storia, edizione 2009</title>
    <link rel='alternate' type='text/html'
      href='http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition'/>
    <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>
    <updated>2009-03-27T21:56:07Z</updated>
    <published>2009-03-27T17:20:42Z</published>
    <category scheme='http://diveintomark.org' term='diveintopython'/>
    <category scheme='http://diveintomark.org' term='docbook'/>
    <category scheme='http://diveintomark.org' term='html'/>
    <summary type='html'>Mettere un intero capitolo in una sola pagina
      sembra eccessivo, ma considerate questo &mdash; finora il mio
      capitolo più lungo equivarrebbe a 75 pagine stampate, e si carica
      in meno di 5 secondi&hellip; Su una connessione in
      dial-up.</summary>
  </entry>
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>L'accessibilità è una padrona inflessibile</title>

```

```

<link rel='alternate' type='text/html'
  href='http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-mistress'/>
<id>tag:diveintomark.org,2009-03-21:/archives/20090321200928</id>
<updated>2009-03-22T01:05:37Z</updated>
<published>2009-03-21T20:09:28Z</published>
<category scheme='http://diveintomark.org' term='accessibility'/>
<summary type='html'>L'ortodossia dell'accessibilità non permette a
  nessuno di mettere in discussione il valore di caratteristiche che
  sono raramente utili e raramente usate.</summary>
</entry>
<entry>
  <author>
    <name>Mark</name>
  </author>
  <title>Una introduzione graduale alla codifica video, parte 1: i formati dei contenitori</title>
  <link rel='alternate' type='text/html'
    href='http://diveintomark.org/archives/2008/12/18/give-part-1-container-formats'/>
  <id>tag:diveintomark.org,2008-12-18:/archives/20081218155422</id>
  <updated>2009-01-11T19:39:22Z</updated>
  <published>2008-12-18T15:54:22Z</published>
  <category scheme='http://diveintomark.org' term='asf'/>
  <category scheme='http://diveintomark.org' term='avi'/>
  <category scheme='http://diveintomark.org' term='encoding'/>
  <category scheme='http://diveintomark.org' term='flv'/>
  <category scheme='http://diveintomark.org' term='GIVE'/>
  <category scheme='http://diveintomark.org' term='mp4'/>
  <category scheme='http://diveintomark.org' term='ogg'/>
  <category scheme='http://diveintomark.org' term='video'/>
  <summary type='html'>Alla fine queste note diventeranno parte di
    una presentazione sulla tecnologia della codifica video.</summary>
</entry>
</feed>

```



12.2. UN CORSO ACCELERATO DI XML IN 5 MINUTI

Se conoscete già XML, potete saltare questa sezione.

XML è un modo generalizzato di descrivere strutture dati gerarchiche. Un *documento* XML contiene uno o più *elementi* che sono delimitati da *tag di apertura* e *chiusura*. Questo è un documento XML completo (anche se noioso):

```
<foo> ①  
</foo> ②
```

1. Questo è il *tag di apertura* dell'elemento `foo`.
2. Questo è il corrispondente *tag di chiusura* dell'elemento `foo`. Allo stesso modo in cui si bilanciano le parentesi nella scrittura, in matematica, o in programmazione, ogni tag di apertura deve essere *terminato* da un corrispondente tag di chiusura.

Gli elementi possono essere *annidati* senza alcun limite di profondità. Un elemento `bar` all'interno di un elemento `foo` si definisce come un *sottoelemento* o un elemento *figlio* di `foo`.

```
<foo>  
  <bar></bar>  
</foo>
```

Il primo elemento di ogni documento XML si chiama *elemento radice*. Un documento XML può avere un solo elemento radice. Il documento che segue **non è un documento XML** perché possiede due elementi radice:

```
<foo></foo>  
<bar></bar>
```

Gli elementi possono essere dotati di *attributi*, che sono coppie nome-valore. Gli attributi sono elencati all'interno del tag di apertura di un elemento e separati da spazi bianchi. I *nomi degli attributi* non possono

essere ripetuti nell'ambito di uno stesso elemento. I *valori degli attributi* devono essere racchiusi tra apici o virgolette.

```
<foo lang='en'> ①  
  <bar id=xml-'papayawhip'>lang="fr"></bar> ②  
</foo>
```

1. L'elemento `foo` possiede un attributo chiamato `lang`. Il valore del suo attributo `lang` è `en`.
2. L'elemento `bar` possiede due attributi chiamati `id` e `lang`. Il valore del suo attributo `lang` è `fr`. Questo non crea conflitti in alcun modo con l'elemento `foo`. Ogni elemento possiede il proprio insieme di attributi.

Nel caso un elemento sia dotato di più di un attributo, l'ordine degli attributi non è significativo. Gli attributi di un elemento formano un insieme non ordinato di chiavi e valori, proprio come un dizionario Python. Non c'è alcun limite sul numero degli attributi che potete definire per ogni elemento.

Gli elementi possono includere *contenuto testuale*.

```
<foo lang='en'>  
  <bar lang='fr'>PapayaWhip</bar>  
</foo>
```

Gli elementi che non contengono testo e non hanno figli si dicono *vuoti*.

```
<foo></foo>
```

Gli elementi vuoti si possono scrivere in maniera abbreviata. Inserendo un carattere `/` nel tag di apertura, potete omettere completamente il tag di chiusura. Il documento XML nell'esempio precedente potrebbe anche essere scritto in questo modo:

```
<foo/>
```

Così come le funzioni Python possono essere dichiarate in *moduli* differenti, gli elementi XML possono essere dichiarati in *spazi di nomi* differenti. Di solito, gli spazi di nomi sono identificati da un URL. Potete usare una

dichiarazione `xmlns` per definire uno *spazio di nomi predefinito*. Una dichiarazione di spazio di nomi ha un aspetto simile a un attributo, ma viene impiegata con uno scopo differente.

```
<feed xmlns='http://www.w3.org/2005/Atom'> ①  
  <title>dive into mark</title> ②  
</feed>
```

1. L'elemento `feed` è nello spazio di nomi `http://www.w3.org/2005/Atom`.
2. Anche l'elemento `title` si trova nello spazio di nomi `http://www.w3.org/2005/Atom`. La dichiarazione dello spazio di nomi ha effetto sull'elemento che la contiene e su tutti i suoi elementi figli.

Potete anche usare una dichiarazione `xmlns:prefisso` per definire uno spazio di nomi e associarlo a un *prefisso*. Dopodiché ogni elemento in quello spazio di nomi dovrà essere esplicitamente dichiarato con il prefisso.

```
<atom:feed xmlns:atom='http://www.w3.org/2005/Atom'> ①  
  <atom:title>dive into mark</atom:title> ②  
</atom:feed>
```

1. L'elemento `feed` è nello spazio di nomi `http://www.w3.org/2005/Atom`.
2. Anche l'elemento `title` si trova nello spazio di nomi `http://www.w3.org/2005/Atom`.

Un qualsiasi riconoscitore XML considera *identici* i due documenti XML precedenti. Spazio di nomi + nome dell'elemento = identità XML. I prefissi esistono solamente per fare riferimento agli spazi di nomi, così l'effettivo nome del prefisso (`atom`) è irrilevante. Gli spazi di nomi corrispondono, i nomi degli elementi corrispondono, gli attributi (o il fatto che siano assenti) corrispondono e il contenuto testuale di ogni elemento corrisponde, quindi i documenti XML sono uguali.

Infine, i documenti XML possono contenere informazioni sulla codifica di carattere nella prima riga, prima dell'elemento radice. (Se siete curiosi di sapere come fa un documento a contenere informazioni che è necessario conoscere prima di poterlo leggere, la Sezione F della specifica XML descrive in dettaglio come risolvere questo Comma 22.)

```
<?xml version='1.0' encoding='utf-8'?>
```


E ora conoscete quel tanto di XML che vi basta per essere pericolosi!



12.3. LA STRUTTURA DI UN FEED ATOM

Pensate a un weblog, o in effetti a qualsiasi sito web il cui contenuto venga frequentemente aggiornato, come CNN.com. Il sito ha un titolo (“CNN.com”), un sottotitolo (“Ultime notizie dagli Stati Uniti e dal mondo, meteo, intrattenimento & servizi filmati”), una data per l’ultimo aggiornamento (“aggiornato alle 12:43 p.m. EDT, sabato 16 maggio 2009”) e una lista di articoli pubblicati a orari differenti. Anche ogni articolo ha un titolo, una data di pubblicazione (e magari anche una data per l’ultimo aggiornamento, se ne è stata pubblicata una modifica oppure è stato corretto un errore di battitura) e un URL unico.

Il formato di syndication Atom è stato progettato per catturare tutte queste informazioni in un formato standard. Il mio weblog e CNN.com hanno un aspetto, un ambito e un pubblico largamente differente, ma entrambi hanno la stessa struttura di base. CNN.com ha un titolo, il mio blog ha un titolo; CNN.com pubblica articoli, io pubblico articoli.

Al livello più alto si trova l’*elemento radice*, che è lo stesso per ogni feed Atom: l’elemento `feed` nello spazio di nomi `http://www.w3.org/2005/Atom`.

```
<feed xmlns='http://www.w3.org/2005/Atom' ①  
      xml:lang='it'> ②
```

1. `http://www.w3.org/2005/Atom` è lo spazio di nomi Atom.
2. Qualsiasi elemento può contenere un attributo `xml:lang` che dichiara la lingua di quell’elemento e dei suoi figli. In questo caso, l’attributo `xml:lang` è dichiarato una sola volta nell’elemento radice, per indicare che l’intero feed è in italiano.

Un feed Atom contiene diversi elementi di informazione sul feed stesso. Questi sono dichiarati come figli dell’elemento radice `feed`.

```

<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'>
  <title>dive into mark</title> ①
  <subtitle>attualmente tra una dipendenza e l'altra</subtitle> ②
  <id>tag:diveintomark.org,2001-07-29:/</id> ③
  <updated>2009-03-27T21:56:07Z</updated> ④
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/> ⑤

```

1. Il titolo di questo feed è dive into mark.
2. Il sottotitolo di questo feed è attualmente tra una dipendenza e l'altra.
3. Ogni feed necessita di un identificatore unico. Leggete la [RFC 4151](#) per sapere come crearne uno.
4. Questo feed è stata aggiornato per l'ultima volta il 27 marzo 2009, alle 21:56 GMT. Di solito, questa data è uguale alla data di ultima modifica dell'articolo più recente.
5. Ora le cose cominciano a diventare interessanti. Questo elemento link non ha contenuto testuale, ma possiede tre attributi: rel, type e href. Il valore di rel vi dice che tipo di collegamento è questo; rel='alternate' significa che è un collegamento a una rappresentazione alternativa per questo feed. L'attributo type='text/html' significa che questo è un collegamento a una pagina HTML. E la destinazione del collegamento viene fornita nell'attributo href.

Ora sappiamo che questo è un feed per un sito chiamato “dive into mark” che è disponibile all'indirizzo <http://diveintomark.org/> e il cui aggiornamento più recente risale al 27 marzo 2009.



L'ordine degli elementi in un feed Atom non è rilevante, sebbene possa esserlo in alcuni documenti XML.

Dopo i metadati a livello di feed troviamo la lista degli articoli più recenti. Un articolo ha un aspetto simile a questo:

```

<entry>

  <author>                                     ①
    <name>Mark</name>
    <uri>http://diveintomark.org/</uri>
  </author>

  <title>Immersione nella storia, edizione 2009</title>      ②

  <link rel='alternate' type='text/html'                ③
    href='http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition' />

  <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>      ④

  <updated>2009-03-27T21:56:07Z</updated>                ⑤

  <published>2009-03-27T17:20:42Z</published>

  <category scheme='http://diveintomark.org' term='diveintopython' />      ⑥
  <category scheme='http://diveintomark.org' term='docbook' />
  <category scheme='http://diveintomark.org' term='html' />

  <summary type='html'>Mettere un intero capitolo in una sola pagina      ⑦
    sembra eccessivo, ma considerate questo &mdash; finora il mio
    capitolo più lungo equivarrebbe a 75 pagine stampate, e si carica
    in meno di 5 secondi&hellip; Su una connessione in
    dial-up.</summary>

</entry>                                             ⑧

```

1. L'elemento `author` vi dice chi ha scritto questo articolo: un certo tizio di nome Mark, che potete trovare a bighegellonare all'indirizzo `http://diveintomark.org/`. (Questo è lo stesso indirizzo del collegamento alternativo contenuto nei metadati del feed, ma non è necessario che lo sia. Molti weblog hanno più di un autore, ognuno col proprio sito personale.)
2. L'elemento `title` vi dà il titolo dell'articolo, "Immersione nella storia, edizione 2009".
3. Come per il collegamento alternativo a livello di feed, questo elemento `link` vi dà l'indirizzo della versione HTML di questo articolo.
4. Ogni voce, come ogni feed, necessita di un identificatore unico.
5. Ogni voce ha due date: la data di pubblicazione (`published`) e la data di ultima modifica (`updated`).
6. Ogni voce può avere un numero arbitrario di categorie. Questo articolo è classificato sotto le categorie `diveintopython`, `docbook` e `html`.

7. L'elemento `summary` fornisce un breve riepilogo dell'articolo. (Esiste anche un elemento `content`, che qui non viene mostrato, da utilizzare se volete includere il testo completo dell'articolo nel vostro feed.) Questo elemento `summary` usa l'attributo `type='html'` specifico di Atom per indicare che questo riepilogo non è in formato di testo semplice, ma è un frammento di HTML. Questo dettaglio è importante, dato che il riepilogo contiene entità specifiche di HTML (— e …) che dovrebbero essere rappresentate come “—” e “...” piuttosto che visualizzate direttamente.
8. Infine, il tag di chiusura per l'elemento `entry` segnala la fine dei metadati per questo articolo.




12.4. RICONOSCERE XML

Python può riconoscere documenti XML in molti modi. Dispone di riconoscitori tradizionali di tipo `DOM` e `SAX`, ma io mi concentrerò su una diversa libreria chiamata `ElementTree`.

```
>>> import xml.etree.ElementTree as etree    ①
>>> tree = etree.parse('esempi/feed.xml')    ②
>>> root = tree.getroot()                    ③
>>> root                                      ④
<Element {http://www.w3.org/2005/Atom}feed at cd1eb0>
```

1. La libreria `ElementTree` fa parte della libreria standard di Python e si trova nel modulo `xml.etree.ElementTree`.
2. Il punto d'ingresso principale della libreria `ElementTree` è la funzione `parse()`, che può prendere come argomento un nome di file o un oggetto simile a un file. Questa funzione riconosce l'intero documento tutto in una volta. Se la memoria disponibile è scarsa, esistono modi per riconoscere un documento XML in maniera incrementale.
3. La funzione `parse()` restituisce un oggetto che rappresenta l'intero documento. Questo oggetto *non* è l'elemento radice. Per ottenere un riferimento all'elemento radice dovete chiamare il metodo `getroot()`.
4. Come vi sareste aspettati, l'elemento radice è l'elemento `feed` nello spazio di nomi `http://www.w3.org/2005/Atom`. La rappresentazione sotto forma di stringa di questo oggetto rafforza un concetto importante: un elemento XML è la combinazione del proprio spazio di nomi e del nome del proprio tag (anche chiamato

nome locale). Ogni elemento in questo documento si trova nello spazio di nomi Atom, quindi l'elemento radice viene rappresentato come {http://www.w3.org/2005/Atom}feed.

 ElementTree rappresenta gli elementi XML come {spaziodinomi}nomelocale. Vedrete e userete questo formato in più punti nella API di ElementTree.

12.4.1. GLI ELEMENTI SONO LISTE

Nella API di ElementTree, un elemento XML si comporta come una lista. Gli elementi della lista sono i figli dell'elemento XML.

```
# continua dall'esempio precedente

>>> root.tag                                ①
'{http://www.w3.org/2005/Atom}feed'

>>> len(root)                                ②
8

>>> for child in root:                        ③
...     print(child)                          ④
...
<Element {http://www.w3.org/2005/Atom}title at e2b5d0>
<Element {http://www.w3.org/2005/Atom}subtitle at e2b4e0>
<Element {http://www.w3.org/2005/Atom}id at e2b6c0>
<Element {http://www.w3.org/2005/Atom}updated at e2b6f0>
<Element {http://www.w3.org/2005/Atom}link at e2b4b0>
<Element {http://www.w3.org/2005/Atom}entry at e2b720>
<Element {http://www.w3.org/2005/Atom}entry at e2b510>
<Element {http://www.w3.org/2005/Atom}entry at e2b750>
```

1. Proseguendo l'esempio precedente, l'elemento radice è {http://www.w3.org/2005/Atom}feed.
2. La “lunghezza” dell'elemento radice è il numero dei suoi elementi figli.
3. Potete usare l'elemento stesso come un iteratore per attraversare tutti i suoi elementi figli.

4. Come potete vedere, ci sono effettivamente 8 elementi figli: tutti quelli che contengono metadati a livello di feed (title, subtitle, id, updated e link) seguiti dai tre elementi entry.

Potreste averlo già indovinato, ma voglio sottolinearlo esplicitamente: la lista degli elementi figli include solamente i figli *diretti*. Ogni elemento entry contiene i propri figli, ma questi non sono inclusi nella lista. Sarebbero inclusi nella lista dei figli di ogni elemento entry, ma non sono inclusi nella lista dei figli dell'elemento feed. Esistono modi di trovare elementi a prescindere da quanto profondamente siano annidati; vedremo due di queste tecniche più avanti in questo capitolo.

12.4.2. GLI ATTRIBUTI SONO DIZIONARI

XML non è semplicemente una collezione di elementi; ogni elemento può anche avere il proprio insieme di attributi. Una volta che avete un riferimento a uno specifico elemento, potete facilmente ottenere i suoi attributi sotto forma di un dizionario Python.

```
# continua dall'esempio precedente

>>> root.attrib                                ①
{'{http://www.w3.org/XML/1998/namespace}lang': 'it'}

>>> root[4]                                    ②
<Element {http://www.w3.org/2005/Atom}link at e181b0>

>>> root[4].attrib                             ③
{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}

>>> root[3]                                    ④
<Element {http://www.w3.org/2005/Atom}updated at e2b4e0>

>>> root[3].attrib                             ⑤
{}
```

1. La proprietà `attrib` è un dizionario degli attributi dell'elemento. Il markup originale qui era `<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'>`. Il prefisso `xml:` si riferisce a uno spazio di nomi predefinito che ogni documento XML può usare senza doverlo dichiarare.
2. Il quinto figlio — `[4]` in una lista i cui indici partono da 0 — è l'elemento `link`.
3. L'elemento `link` ha tre attributi: `href`, `type` e `rel`.

4. Il quarto figlio — [3] in una lista i cui indici partono da 0 — è l'elemento `updated`.
5. L'elemento `updated` non ha attributi, quindi `attrib` in questo caso è semplicemente un dizionario vuoto.



12.5. CERCARE NODI ALL'INTERNO DI UN DOCUMENTO XML

Finora abbiamo lavorato con questo documento XML in maniera “top down”, dall'alto verso il basso, partendo dall'elemento radice, recuperando i suoi elementi figli, e così via attraverso tutto il documento. Ma molti usi di XML vi richiedono di trovare elementi specifici. `ElementTree` può fare anche questo.

```
>>> import xml.etree.ElementTree as etree
>>> tree = etree.parse('esempi/feed.xml')
>>> root = tree.getroot()
>>> root.findall('{http://www.w3.org/2005/Atom}entry') ①
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
>>> root.tag
'{http://www.w3.org/2005/Atom}feed'
>>> root.findall('{http://www.w3.org/2005/Atom}feed') ②
[]
>>> root.findall('{http://www.w3.org/2005/Atom}author') ③
[]
```

1. Il metodo `findall()` trova elementi figli che corrispondono a una richiesta specifica. (I dettagli sul formato della richiesta arriveranno in un minuto.)
2. Tutti gli elementi — incluso l'elemento radice, ma anche gli elementi figli — hanno un metodo `findall()`. Il metodo trova tutti gli elementi che corrispondono alla richiesta cercando tra gli elementi figli. Ma perché in questo caso non viene trovato alcun risultato? Sebbene possa non apparire immediato, questa particolare richiesta cerca solo tra i figli di un elemento. Dato che l'elemento radice `feed` non ha figli chiamati `feed`, questa richiesta restituisce una lista vuota.

3. Anche questo risultato potrebbe sorprendervi. C'è un elemento author in questo documento, anzi, in effetti ce ne sono tre (uno per ogni entry). Ma quegli elementi author non sono *figli diretti* dell'elemento radice, bensì “nipoti” (letteralmente, un elemento figlio di un elemento figlio). Se volete cercare elementi author a qualsiasi livello di profondità potete farlo, ma la forma della richiesta è leggermente differente.

```
>>> tree.findall('{http://www.w3.org/2005/Atom}entry')    ①
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]

>>> tree.findall('{http://www.w3.org/2005/Atom}author')    ②
[]
```

1. Per convenienza, l'oggetto `tree` (restituito dalla funzione `etree.parse()`) possiede diversi metodi speculari a quelli dell'elemento radice. I risultati di questi metodi sono gli stessi che avreste ottenuto invocando il metodo `tree.getroot().findall()`.
2. In maniera forse sorprendente, questa richiesta non trova gli elementi `author` in questo documento. Perché no? Perché questa è solo una scorciatoia per `tree.getroot().findall('{http://www.w3.org/2005/Atom}author')`, che significa “trova tutti gli elementi `author` che sono figli dell'elemento radice”. Gli elementi `author` non sono figli dell'elemento radice, ma sono figli degli elementi `entry`. Quindi la richiesta non restituisce alcuna corrispondenza.

Esiste anche un metodo `find()` che restituisce il primo elemento corrispondente. Questo è utile per situazioni in cui vi aspettate una sola corrispondenza o vi interessa solo la prima corrispondenza nel caso ce ne sia più di una.


```

>>> entries = tree.findall('{http://www.w3.org/2005/Atom}entry') ①
>>> len(entries)
3
>>> title_element = entries[0].find('{http://www.w3.org/2005/Atom}title') ②
>>> title_element.text
'Immersione nella storia, edizione 2009'
>>> foo_element = entries[0].find('{http://www.w3.org/2005/Atom}foo') ③
>>> foo_element
>>> type(foo_element)
<class 'NoneType'>

```

1. Avete visto questo metodo in azione nell'esempio precedente. Trova tutti gli elementi `atom:entry`.
2. Il metodo `find()` accetta una richiesta `ElementTree` e restituisce il primo elemento corrispondente.
3. In questa voce non ci sono elementi chiamati `foo`, quindi questa richiesta restituisce `None`.

☞ C'è una sgradita “sorpresa” che il metodo `find()` finirà per riservarvi. In un contesto logico, gli oggetti elemento di `ElementTree` verranno valutati come `False` se non contengono figli (cioè se `len(element)` vale 0). Questo significa che `if element.find('...')` non sta verificando che il metodo `find()` abbia trovato un elemento corrispondente, ma sta verificando che quell'elemento corrispondente abbia elementi figli! Per verificare che il metodo `find()` abbia restituito un elemento usate `if element.find('...') is not None`.

Esiste un modo per cercare gli elementi *discendenti*, cioè figli, nipoti e qualsiasi elemento a qualsiasi livello di profondità.

```

>>> all_links = tree.findall('//{http://www.w3.org/2005/Atom}link') ①
>>> all_links
[<Element {http://www.w3.org/2005/Atom}link at e181b0>,
 <Element {http://www.w3.org/2005/Atom}link at e2b570>,
 <Element {http://www.w3.org/2005/Atom}link at e2b480>,
 <Element {http://www.w3.org/2005/Atom}link at e2b5a0>]
>>> all_links[0].attrib ②
{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[1].attrib ③
{'href': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[2].attrib
{'href': 'http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-mistress',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[3].attrib
{'href': 'http://diveintomark.org/archives/2008/12/18/give-part-1-container-formats',
 'type': 'text/html',
 'rel': 'alternate'}

```

1. Questa richiesta — `//{http://www.w3.org/2005/Atom}link` — è molto simile agli esempi precedenti, tranne per i due caratteri di slash che si trovano all’inizio della richiesta. Quei due slash significano “non cercare solo tra i figli diretti, voglio *qualsiasi* elemento a prescindere dal livello di profondità”. Quindi il risultato è una lista che contiene quattro elementi `link`, non uno solo.
2. Il primo risultato è un figlio diretto dell’elemento radice. Come potete vedere dai suoi attributi, quello è il collegamento alternativo a livello di feed che punta alla versione HTML del sito web descritto dal feed.
3. Gli altri tre risultati sono i collegamenti alternativi a livello di singola voce. Ogni elemento `entry` ha un singolo elemento figlio `link` e, a causa del doppio slash all’inizio della richiesta, questa richiesta li trova tutti e tre.

Nel complesso, il metodo `findall()` è una caratteristica molto potente di `ElementTree`, ma la sintassi utilizzata nelle sue richieste può risultare un po' sorprendente. Viene ufficialmente descritta come un "sottoinsieme ristretto delle espressioni XPath". XPath è uno standard W3C per effettuare ricerche nei documenti XML. La sintassi delle richieste di `ElementTree` è abbastanza simile a XPath da consentire ricerche di base, ma abbastanza diversa da potervi infastidire se conoscete già XPath. Ora diamo un'occhiata a una libreria XML di terze parti che estende la API di `ElementTree` con un supporto completo per XPath.



12.6. PROSEGUIRE CON LXML

`lxml` è una libreria open source di terze parti basata sul popolare riconoscitore `libxml2`. Fornisce una API compatibile al 100% con `ElementTree`, poi la estende con un supporto completo per XPath 1.0 e alcune altre raffinatezze. Ne esistono pacchetti di installazione disponibili per Windows, mentre gli utenti Linux dovrebbero sempre provare a usare strumenti specifici per la loro distribuzione, come `yum` o `apt-get`, che permettono di installare librerie precompilate scaricandole dagli archivi della distribuzione. Altrimenti, dovrete installare `lxml` manualmente.

```
>>> from lxml import etree ①
>>> tree = etree.parse('esempi/feed.xml') ②
>>> root = tree.getroot() ③
>>> root.findall('{http://www.w3.org/2005/Atom}entry') ④
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
```

1. Una volta importata, `lxml` fornisce la stessa API della libreria built-in `ElementTree`.
2. La funzione `parse()`: la stessa di `ElementTree`.
3. Il metodo `getroot()`: anche questo è lo stesso.
4. Il metodo `findall()`: esattamente identico.

Per documenti XML di grandi dimensioni, `lxml` è significativamente più veloce rispetto alla libreria built-in `ElementTree`. Se state usando solo la API di `ElementTree` e volete sfruttare l'implementazione più veloce disponibile, potete provare a importare `lxml` e ricorrere all'alternativa built-in `ElementTree` solo nel caso in cui `lxml` non sia presente.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

Ma `lxml` è molto più di una `ElementTree` più veloce. Il suo metodo `findall()` include il supporto per espressioni più complicate.

```
>>> import lxml.etree ①
>>> tree = lxml.etree.parse('esempi/feed.xml')
>>> tree.findall('://{http://www.w3.org/2005/Atom}*[@href]') ②
[<Element {http://www.w3.org/2005/Atom}link at eeb8a0>,
 <Element {http://www.w3.org/2005/Atom}link at eeb990>,
 <Element {http://www.w3.org/2005/Atom}link at eeb960>,
 <Element {http://www.w3.org/2005/Atom}link at eeb9c0>]
>>> tree.findall("/{http://www.w3.org/2005/Atom}*[@href='http://diveintomark.org/']") ③
[<Element {http://www.w3.org/2005/Atom}link at eeb930>]
>>> NS = '{http://www.w3.org/2005/Atom}'
>>> tree.findall('://{NS}author[{NS}uri]'.format(NS=NS)) ④
[<Element {http://www.w3.org/2005/Atom}author at ee8a80>,
 <Element {http://www.w3.org/2005/Atom}author at ee8ba0>]
```

1. In questo esempio, scriverò `import lxml.etree` (invece di scrivere, diciamo, `from lxml import etree`) per enfatizzare che queste caratteristiche sono specifiche di `lxml`.
2. Questa richiesta trova tutti gli elementi nello spazio di nomi `Atom`, ovunque nel documento, che abbiano un attributo `href`. I caratteri `//` all'inizio della richiesta significano “elementi ovunque (non solo come figli dell'elemento radice)”. `{http://www.w3.org/2005/Atom}` significa “solo elementi nello spazio di nomi `Atom`”. `*` significa “elementi con un qualsiasi nome locale”. E `[@href]` significa “ha un attributo `href`”.
3. La richiesta trova tutti gli elementi `Atom` con un attributo `href` il cui valore sia `http://diveintomark.org/`.

4. Dopo aver eseguito alcune rapide formattazioni di stringhe (perché altrimenti le richieste composte diventano ridicolmente lunghe), questa richiesta cerca gli elementi Atom author che hanno un elemento Atom uri come figlio. Questa richiesta restituisce solo due elementi author, quelli nella prima e nella seconda entry. L'elemento author nell'ultima entry contiene solo un figlio name, non un figlio uri.

Non vi basta? lxml include anche il supporto per espressioni XPath 1.0 arbitrarie. Non cercherò di approfondire la sintassi XPath (ci si potrebbe scrivere un intero libro!) ma vi mostrerò come si integra in lxml.

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('esempi/feed.xml')
>>> NSMAP = {'atom': 'http://www.w3.org/2005/Atom'} ①
>>> entries = tree.xpath("//atom:category[@term='accessibility']/..", ②
...     namespaces=NSMAP)
>>> entries ③
[<Element {http://www.w3.org/2005/Atom}entry at e2b630>]
>>> entry = entries[0]
>>> entry.xpath('./atom:title/text()', namespaces=NSMAP) ④
["L'accessibilità è una padrona inflessibile"]
```

1. Per effettuare richieste XPath su elementi in uno spazio di nomi, avete bisogno di definire una corrispondenza tra prefissi e spazi di nomi sotto la semplice forma di un dizionario Python.
2. Ecco una richiesta XPath. L'espressione XPath cerca gli elementi category (nello spazio di nomi Atom) che contengono un attributo term con il valore accessibility. Ma questo non è l'effettivo risultato della richiesta. Guardate alla fine della stringa di richiesta: avete notato quel /..? Questo significa "e poi restituisci l'elemento genitore dell'elemento category che hai appena trovato". Così questa singola richiesta XPath troverà tutte le voci con un elemento figlio <category term='accessibility'>.
3. La funzione xpath() restituisce una lista di oggetti ElementTree. In questo documento c'è solo una voce con un elemento category il cui attributo term abbia il valore accessibility.
4. Le espressioni XPath non restituiscono sempre una lista di elementi. Tecnicamente, la rappresentazione DOM di un documento XML non contiene elementi, ma contiene *nodi*. A seconda del loro tipo, i nodi possono essere elementi, attributi, o anche contenuti testuali. Il risultato di una richiesta XPath è una lista di nodi. Questa richiesta restituisce una lista di nodi di testo: il contenuto testuale (text()) dell'elemento title (atom:title) che è un figlio dell'elemento corrente (.).



12.7. GENERARE XML

Il supporto di Python per XML non si limita al riconoscimento di documenti esistenti. Potete anche creare documenti XML da zero.

```
>>> import xml.etree.ElementTree as etree

>>> new_feed = etree.Element('{http://www.w3.org/2005/Atom}feed',    ①
...     attrib={'{http://www.w3.org/XML/1998/namespace}lang': 'it'})  ②
>>> print(etree.tostring(new_feed))                                     ③

<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='it'/>
```

1. Per creare un nuovo elemento dovete istanziare la classe `Element`. Passate il nome dell'elemento (spazio di nomi + nome locale) come primo argomento. Questa istruzione crea un elemento `feed` nello spazio di nomi `Atom`. Questo sarà l'elemento radice del nostro nuovo documento.
2. Per aggiungere attributi all'elemento appena creato, passate un dizionario di nomi e valori di attributi come argomento `attrib`. Notate che i nomi di attributi dovrebbero essere nel formato standard di `ElementTree` `{spaziodinomi}nomelocale`.
3. Potete serializzare qualsiasi elemento (e i suoi figli) in ogni momento utilizzando la funzione `tostring()` di `ElementTree`.

Quella serializzazione vi ha sorpreso? Il modo in cui `ElementTree` serializza gli spazi di nomi XML è tecnicamente accurato ma non ottimale. Il documento XML di esempio all'inizio di questo capitolo definiva uno *spazio di nomi predefinito* (`xmlns='http://www.w3.org/2005/Atom'`). Definire uno spazio di nomi predefinito è utile per documenti — come i feed Atom — dove tutti gli elementi sono nello stesso spazio di nomi, perché potete dichiarare lo spazio di nomi una volta sola e dichiarare ogni elemento utilizzando semplicemente il suo nome locale (`<feed>`, `<link>`, `<entry>`). Non c'è alcun bisogno di usare un prefisso a meno che non vogliate dichiarare elementi appartenenti a un altro spazio di nomi.

Un riconoscitore XML non sarà in grado di “vedere” alcuna differenza tra un documento XML con uno spazio di nomi predefinito e un documento XML con uno spazio di nomi che utilizza un prefisso. La rappresentazione DOM che risulta da questa serializzazione:

```
<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='it'/>
```

è identica alla rappresentazione DOM di questa serializzazione:

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'/>
```

L'unica differenza concreta è che la seconda serializzazione è più corta di alcuni caratteri. Se dovessimo riscrivere interamente il nostro feed di esempio con un prefisso `ns0:` in ogni tag di apertura e chiusura, aggiungerei 4 caratteri per tag di apertura \times 79 tag + 4 caratteri per la dichiarazione di spazio di nomi, per un totale di 320 caratteri. Assumendo una codifica UTF-8, il totale corrisponde a 320 byte aggiuntivi. (Dopo aver compresso il documento tramite `gzip`, la differenza scende a 21 byte, ciò nondimeno 21 byte sono 21 byte.) Forse per voi non ha importanza, ma per qualcosa come un feed Atom, che potrebbe essere scaricato diverse migliaia di volte a ogni cambiamento, il risparmio di alcuni byte per ogni richiesta può accumularsi velocemente.

La libreria built-in `ElementTree` non offre un controllo così accurato sulla serializzazione di elementi in uno spazio di nomi, ma `lxml` lo fa.

```
>>> import lxml.etree

>>> NSMAP = {None: 'http://www.w3.org/2005/Atom'} ①

>>> new_feed = lxml.etree.Element('feed', nsmap=NSMAP) ②

>>> print(lxml.etree.tounicode(new_feed)) ③

<feed xmlns='http://www.w3.org/2005/Atom'/>

>>> new_feed.set('{http://www.w3.org/XML/1998/namespace}lang', 'it') ④

>>> print(lxml.etree.tounicode(new_feed))

<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'/>
```

1. Per cominciare, definite una corrispondenza di spazi di nomi sotto forma di un dizionario in cui i valori siano gli spazi di nomi e le chiavi siano i prefissi desiderati. Usate `None` come prefisso per dichiarare uno spazio di nomi predefinito.
2. Ora potete passare l'argomento `nsmap` specifico per `lxml` quando create un elemento, così `lxml` rispetterà i prefissi per gli spazi di nomi che avete definito.
3. Come vi sareste aspettati, questa serializzazione definisce lo spazio di nomi Atom come spazio di nomi predefinito e dichiara l'elemento `feed` senza un prefisso per lo spazio di nomi.


4. Oops, abbiamo dimenticato di aggiungere l'attributo `xml:lang`. Potete sempre aggiungere attributi a qualsiasi elemento tramite il metodo `set()`. Questo metodo prende due argomenti: il nome dell'attributo nel formato standard di `ElementTree`, poi il valore dell'attributo. (Questo metodo non è specifico per `lxml`. L'unica parte specifica per `lxml` in questo esempio era l'argomento `nsmap` usato per controllare i prefissi degli spazi di nomi nell'uscita serializzata.)

I documenti XML si limitano ad avere un elemento per documento? No, naturalmente no. Potete facilmente creare anche elementi figli.

```
>>> title = lxml.etree.SubElement(new_feed, 'title',           ①
...     attrib={'type':'html'})                                ②
>>> print(lxml.etree.tounicode(new_feed))                      ③
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'><title type='html' /></feed>
>>> title.text = 'dive into &hellip;'                          ④
>>> print(lxml.etree.tounicode(new_feed))                      ⑤
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'><title type='html'>dive into &amp;hellip;</tit
>>> print(lxml.etree.tounicode(new_feed, pretty_print=True))    ⑥
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'>
  <title type='html'>dive into &amp;hellip;</title>
</feed>
```

1. Per creare un elemento figlio di un elemento esistente, istanziate la classe `SubElement`. Gli unici argomenti richiesti sono l'elemento genitore (`new_feed` in questo caso) e il nome del nuovo elemento. Dato che questo elemento figlio eredita la corrispondenza di spazi di nomi del suo genitore, qui non c'è bisogno di dichiarare nuovamente lo spazio di nomi o il prefisso.
2. Potete anche passare in ingresso un dizionario di attributi in cui le chiavi siano i nomi degli attributi e i valori siano i valori degli attributi.
3. Come vi sareste aspettati, il nuovo elemento `title` è stato creato nello spazio di nomi `Atom` e inserito come figlio dell'elemento `feed`. Dato che l'elemento `title` non ha contenuto testuale e non ha figli propri, `lxml` lo serializza come un elemento vuoto (utilizzando l'abbreviazione `/>`).
4. Per impostare il contenuto testuale di un elemento, impostate semplicemente la sua proprietà `text`.
5. Ora l'elemento `title` viene serializzato insieme al suo contenuto testuale. Durante la serializzazione è necessario effettuare l'escape del contenuto testuale che contiene segni di minore oppure caratteri di E commerciale. `lxml` gestisce questo processo in maniera automatica.

6. Potete anche utilizzare l'argomento con nome `pretty_print` per ottenere una stampa formattata del testo serializzato, nella quale viene inserito un ritorno a capo dopo i tag di chiusura e prima dei tag di apertura di elementi che contengono elementi figli ma non includono contenuto testuale. In termini tecnici, `lxml` aggiunge “spazio bianco non significativo” per rendere il risultato più leggibile.

 Potreste anche voler provare `xmlwitch`, un'altra libreria di terze parti per generare XML che fa largo uso della istruzione `with` per rendere più leggibile il codice di generazione di documenti XML.



12.8. RICONOSCERE DOCUMENTI XML CONTENENTI ERRORI DI MALFORMAZIONE

La specifica XML obbliga tutti i riconoscitori XML conformanti a impiegare una “gestione degli errori draconiana”. Vale a dire che i riconoscitori si devono “fermare e prendere fuoco” non appena scoprono un qualsiasi tipo di errore di malformazione nel documento XML. Gli errori di malformazione includono tag di apertura e chiusura che non corrispondono, entità non definite, caratteri Unicode illegali e un certo numero di altre regole esoteriche. Questo è in netto contrasto con altri formati comuni come HTML — il vostro browser non smette di visualizzare una pagina web se dimenticate di chiudere un tag HTML o di effettuare l'escape di una E commerciale nel valore di un attributo. (È un equivoco comune che HTML non abbia una gestione degli errori definita. La gestione degli errori HTML è in realtà piuttosto ben definita, ma è significativamente più complicata di “fermati e prendi fuoco al primo errore”).

Alcune persone (me compreso) credono che l'obbligo a una gestione degli errori draconiana sia stato un errore da parte degli inventori di XML. Non fraintendetemi, posso certamente vedere il fascino di una semplificazione delle regole di gestione degli errori. Ma nella pratica il concetto di “malformazione” è più complicato di quanto sembra, specialmente per documenti XML (come i feed Atom) che sono pubblicati sul web e serviti attraverso HTTP. Nonostante la maturità di XML, che si è assestato sulla gestione draconiana degli errori nel 1997, i rilevamenti mostrano continuamente che una frazione significativa di feed Atom sul web è afflitta da errori di malformazione.

Quindi, ho ragioni sia teoriche che pratiche per riconoscere documenti XML “a tutti i costi”, cioè per *non* fermarmi e prendere fuoco al primo errore di malformazione. Se vi trovate a voler fare questa cosa anche voi, lxml può aiutarvi.

Ecco un frammento di un documento XML difettoso. Ho evidenziato l'errore di malformazione.

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'>
  <title>dive into &hellip;</title>
  ...
</feed>
```

Quello è un errore perché l'entità … non è definita in XML. (È definita in HTML.) Se provate a riconoscere questo feed difettoso con le impostazioni di default, lxml si bloccherà sull'entità non definita.

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('esempi/feed-broken.xml')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lxml.etree.pyx", line 2693, in lxml.etree.parse (src/lxml/lxml.etree.c:52591)
  File "parser.pxi", line 1478, in lxml.etree._parseDocument (src/lxml/lxml.etree.c:75665)
  File "parser.pxi", line 1507, in lxml.etree._parseDocumentFromURL (src/lxml/lxml.etree.c:75993)
  File "parser.pxi", line 1407, in lxml.etree._parseDocFromFile (src/lxml/lxml.etree.c:75002)
  File "parser.pxi", line 965, in lxml.etree._BaseParser._parseDocFromFile (src/lxml/lxml.etree.c:7202)
  File "parser.pxi", line 539, in lxml.etree._ParserContext._handleParseResultDoc (src/lxml/lxml.etree.c:68877)
  File "parser.pxi", line 625, in lxml.etree._handleParseResult (src/lxml/lxml.etree.c:68877)
  File "parser.pxi", line 565, in lxml.etree._raiseParseError (src/lxml/lxml.etree.c:68125)
lxml.etree.XMLSyntaxError: Entity 'hellip' not defined, line 3, column 28
```

Per riconoscere questo documento XML difettoso nonostante il suo errore di malformazione, avete bisogno di creare un riconoscitore XML personalizzato.

```

>>> parser = lxml.etree.XMLParser(recover=True) ①
>>> tree = lxml.etree.parse('esempi/feed-broken.xml', parser) ②
>>> parser.error_log ③
esempi/feed-broken.xml:3:28:FATAL:PARSER:ERR_UNDECLARED_ENTITY: Entity 'hellip' not defined
>>> tree.findall('{http://www.w3.org/2005/Atom}title')
[<Element {http://www.w3.org/2005/Atom}title at ead510>]
>>> title = tree.findall('{http://www.w3.org/2005/Atom}title')[0]
>>> title.text ④
'dive into '
>>> print(lxml.etree.tounicode(tree.getroot())) ⑤
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'>
  <title>dive into </title>
  .
  . [il resto della serializzazione viene omissso per brevità]
  .

```

1. Per creare un riconoscitore personalizzato, istanziate la classe `lxml.etree.XMLParser`. Questa classe può prendere un certo numero di differenti argomenti con nome. Quello a cui siamo interessati qui è l'argomento `recover`. Quando questo argomento viene impostato a `True`, il riconoscitore XML farà del suo meglio per “compensare” gli errori di malformazione.
2. Per riconoscere un documento XML utilizzando il vostro riconoscitore personalizzato, passate l'oggetto `parser` come secondo argomento alla funzione `parse()`. Notate che ora `lxml` non solleva alcuna eccezione relativa all'entità `…` non definita.
3. Il riconoscitore mantiene un registro degli errori di malformazione che ha incontrato. (In effetti, questo è vero a prescindere dal fatto di averlo impostato per compensare quegli errori oppure no.)
4. Dato che non sapeva cosa fare con l'entità `…` non definita, il riconoscitore l'ha semplicemente scartata in silenzio. Il contenuto testuale dell'elemento `title` diventa `'dive into '`.
5. Come potete vedere dalla serializzazione, l'entità `…` non è stata spostata, ma è stata semplicemente scartata.

È importante sottolineare che **non c'è alcuna garanzia di interoperabilità** quando si utilizzano riconoscitori XML che “compensano” gli errori di malformazione. Un riconoscitore differente potrebbe decidere che ha riconosciuto l'entità `…` da HTML, rimpiazzandola poi con `…`. Questa soluzione è “migliore”? Forse. È “più corretta”? No, entrambi i riconoscitori sbagliano allo stesso modo. Il

comportamento corretto (in accordo con la specifica XML) è di fermarsi e prendere fuoco. Se avete deciso di non farlo, siete abbandonati a voi stessi.



12.9. LETTURE DI APPROFONDIMENTO

- [XML su Wikipedia.org](#)
- [La API XML di ElementTree](#)
- [Elementi e alberi di elementi](#)
- [Il supporto XPath in ElementTree](#)
- [La funzione iterparse di ElementTree](#)
- [lxml](#)
- [Riconoscere XML e HTML con lxml](#)
- [Usare XPath e XSLT con lxml](#)
- [xmlwitch](#)

CAPITOLO 13. SERIALIZZARE OGGETTI PYTHON

“ Da quando viviamo in questo appartamento, ogni sabato mi sono alzato alle 6:15, mi sono preparato una tazza di cereali con 1/16 di litro di latte parzialmente scremato, mi sono seduto su questo lato di questo divano, ho acceso la TV su BBC America e ho guardato Doctor Who. ”
— Sheldon, *The Big Bang Theory*

13.1. IMMERSIONE!

Quando lo analizzate in superficie, il concetto di serializzazione è semplice. Avete in memoria una struttura dati che volete salvare, riutilizzare, o inviare a qualcun altro. Come fareste? Be', questo dipende da come volete salvarla, dal modo in cui pensate di riutilizzarla e dal destinatario a cui desiderate inviarla. Molti videogiochi vi permettono di salvare i vostri progressi quando uscite dal gioco e di ricominciare da dove eravate rimasti quando rientrate nel gioco. (In realtà, anche molte applicazioni che non sono giochi lo fanno.) In questo caso, una struttura dati che cattura “i vostri progressi finora” deve essere memorizzata su disco quando uscite, poi caricata dal disco quando rientrate. I dati sono pensati solo per essere usati dallo stesso programma che li ha creati, mai per venire inviati in rete né per essere letti da altri programmi che non siano quello che li ha creati. Quindi, l'interoperabilità si limita a garantire che versioni più recenti del programma possano leggere i dati salvati dalle versioni precedenti.

Per casi come questi, il modulo `pickle` è l'ideale: fa parte della libreria standard di Python, quindi è sempre disponibile; è veloce, perché la maggior parte del modulo è scritta in C, come lo stesso interprete Python; e può memorizzare strutture dati Python arbitrariamente complesse.

Cosa può memorizzare il modulo `pickle`?

- Tutti i tipi di dato nativi supportati da Python: booleani, interi, numeri in virgola mobile, numeri complessi, stringhe, oggetti `bytes`, array di `byte` e `None`.
- Liste, tuple, dizionari e insiemi contenenti qualsiasi combinazione di tipi di dato nativi.

- Liste, tuple, dizionari e insiemi contenenti qualsiasi combinazione di liste, tuple, dizionari e insiemi contenenti qualsiasi combinazione di tipi di dato nativi (e così via, fino al massimo livello di annidamento supportato da Python).
- Funzioni, classi e istanze di classi (con alcune avvertenze).

Se questo non vi basta, potete anche estendere il modulo `pickle`. Se siete interessati alla estendibilità, controllate le letture di approfondimento al termine del capitolo.

13.1.1. UNA NOTA RAPIDA SUGLI ESEMPI DI QUESTO CAPITOLO

Questo capitolo narra il proprio racconto usando due Shell Python. Tutti gli esempi in questo capitolo fanno parte dell'arco di una singola storia. Vi verrà chiesto di spostarvi avanti e indietro tra le due Shell Python man mano che vi mostro le funzioni dei moduli `pickle` e `json`.

Per evitare di confondervi, aprite una Shell Python e definite la seguente variabile:

```
>>> shell = 1
```

Tenete aperta quella finestra. Ora aprite un'altra Shell Python e definite la seguente variabile:

```
>>> shell = 2
```

In tutto questo capitolo, userò la variabile `shell` per indicare la Shell Python che viene usata in ogni esempio.



13.2. SALVARE DATI IN UN FILE PICKLE

Il modulo `pickle` lavora con le strutture dati. Costruiamone una.

```

>>> shell
1
>>> entry = {}
>>> entry['title'] = 'Immersione nella storia, edizione 2009'
>>> entry['article_link'] = 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition'
>>> entry['comments_link'] = None
>>> entry['internal_id'] = b'\xDE\xD5\xB4\xF8'
>>> entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> entry['published'] = True
>>> import time
>>> entry['published_date'] = time.strptime('Fri Mar 27 22:20:42 2009')
>>> entry['published_date']
time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4, tm_ye

```

1. Seguitemi nella Shell Python n° I.
2. L'idea qui è quella di costruire un dizionario Python che possa rappresentare qualcosa di utile, come una voce in un feed Atom. Ma voglio anche assicurarmi che contenga diversi tipi di dato, per mettere in risalto il modulo pickle. Questi valori non hanno alcun significato particolare.
3. Il modulo time contiene una struttura dati (time_struct) per rappresentare un punto nel tempo (accurato al millisecondo) e alcune funzioni per manipolare queste strutture. La funzione strptime() prende una stringa formattata e la converte in un oggetto time_struct. Il formato di questa stringa è quello predefinito, ma potete controllarlo con i codici di formato. Leggete la documentazione sul modulo time per maggiori dettagli.

Questo sembra proprio un bel dizionario Python. Salviamolo in un file.

```

>>> shell ①
1
>>> import pickle
>>> with open('entry.pickle', 'wb') as f: ②
...     pickle.dump(entry, f) ③
...

```

1. Ci troviamo ancora nella Shell Python n° I.

2. Usiamo la funzione `open()` per aprire il file, impostando la modalità a `'wb'` in modo da aprirlo in scrittura in modalità binaria. Circondiamo la funzione con una istruzione `with` per avere la garanzia che il file venga chiuso automaticamente quando abbiamo finito di lavorare.
3. La funzione `dump()` del modulo `pickle` prende una struttura dati Python serializzabile, la serializza in un formato binario specifico per Python usando la versione più recente del protocollo pickle e ne salva la forma serializzata in un file aperto.

Quest'ultima frase è molto importante.

- Il modulo `pickle` prende una struttura dati Python e la salva in un file.
- Per fare questo, *serializza* la struttura dati usando un formato di dati chiamato “protocollo pickle”.
- Il protocollo pickle è specifico per Python; non c'è alcuna garanzia di compatibilità verso altri linguaggi. Probabilmente non potreste prendere il file `entry.pickle` che avete appena creato e farci qualcosa di utile in Perl, PHP, Java, o qualsiasi altro linguaggio.
- Non tutte le strutture dati Python possono essere serializzate dal modulo `pickle`. Il protocollo pickle è cambiato diverse volte man mano che nuovi tipi di dato sono stati aggiunti al linguaggio Python, ma ci sono ancora alcune restrizioni.
- Come risultato di questi cambiamenti, non c'è alcuna garanzia di compatibilità tra diverse versioni di Python. Le versioni più recenti supportano i formati di serializzazione più vecchi, ma le vecchie versioni di Python non supportano i nuovi formati (dato che non supportano i nuovi tipi di dato).
- A meno che non specifichiate diversamente, le funzioni del modulo `pickle` useranno la versione più recente del protocollo pickle. Questo vi garantisce la massima flessibilità nei tipi di dato che potete serializzare, ma significa anche che il file risultante non potrà essere letto da vecchie versioni di Python che non supportano la versione più recente del protocollo pickle.
- L'ultima versione del protocollo pickle è un formato binario. Assicuratevi di aprire i vostri file pickle in modalità binaria, o i dati verranno rovinati durante la scrittura.



13.3. CARICARE DATI DA UN FILE PICKLE

Ora spostatevi nella seconda Shell Python — cioè quella in cui *non* avete creato il dizionario `entry`.


```

>>> shell ①
2
>>> entry ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import pickle
>>> with open('entry.pickle', 'rb') as f: ③
...     entry = pickle.load(f) ④
...
>>> entry ⑤
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Immersione nella storia, edizione 2009',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link':
 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20, tm_sec=
 'published': True}

```

1. Questa è la Shell Python n°2.
2. Non c'è alcuna variabile entry definita qui. Avete definito una variabile entry nella Shell Python n°1, ma questo è un ambiente con un proprio stato completamente differente.
3. Aprite il file entry.pickle che avete creato nella Shell Python n°1. Il modulo pickle usa un formato di dati binario, quindi dovrete sempre aprire i file pickle in modalità binaria.
4. La funzione pickle.load() prende un oggetto stream, legge i dati serializzati dal flusso, crea un nuovo oggetto Python, ricrea i dati serializzati nel nuovo oggetto Python e infine restituisce questo nuovo oggetto.
5. Ora la variabile entry è un dizionario contenente chiavi e valori che sembrano familiari.

Il ciclo pickle.dump() / pickle.load() dà come risultato una nuova struttura dati che è uguale alla struttura dati originale.

```

>>> shell ①
1
>>> with open('entry.pickle', 'rb') as f: ②
...     entry2 = pickle.load(f) ③
...
>>> entry2 == entry ④
True
>>> entry2 is entry ⑤
False
>>> entry2['tags'] ⑥
('diveintopython', 'docbook', 'html')
>>> entry2['internal_id']
b'\xDE\xD5\xB4\xF8'

```

1. Tornate indietro alla Shell Python n°1.
2. Aprite il file `entry.pickle`.
3. Caricate i dati serializzati in una nuova variabile chiamata `entry2`.
4. Python conferma che i due dizionari `entry` ed `entry2` sono uguali. In questa shell avete costruito `entry` da zero, cominciando con un dizionario vuoto e assegnando manualmente i valori a chiavi specifiche. Avete serializzato questo dizionario memorizzandolo nel file `entry.pickle`. Ora avete letto i dati serializzati da quel file e avete creato una riproduzione perfetta della struttura dati originale.
5. L'uguaglianza non è la stessa cosa dell'identità. Ho detto che avete creato una *riproduzione perfetta* della struttura dati originale, ed è vero, ma è ancora una copia.
6. Per ragioni che diventeranno chiare più avanti in questo capitolo, voglio sottolineare che il valore della chiave `'tags'` è una tupla e che il valore della chiave `'internal_id'` è un oggetto bytes.

*
**

13.4. SERIALIZZARE SENZA UN FILE

Gli esempi nelle sezioni precedenti hanno mostrato come serializzare un oggetto Python direttamente in un file su disco. E se voi non voleste o non aveste bisogno di un file? Potete anche serializzare in un oggetto bytes in memoria.

```
>>> shell
1
>>> b = pickle.dumps(entry)      ①
>>> type(b)                      ②
<class 'bytes'>
>>> entry3 = pickle.loads(b)     ③
>>> entry3 == entry              ④
True
```

1. La funzione `pickle.dumps()` (notate la 's' alla fine del nome della funzione) effettua la stessa serializzazione della funzione `pickle.dump()`, ma invece di prendere un oggetto stream e scrivere i dati serializzati in un file su disco restituisce semplicemente i dati serializzati.
2. Dato che il protocollo pickle usa un formato di dati binario, la funzione `pickle.dumps()` restituisce un oggetto bytes.
3. La funzione `pickle.loads()` (anche in questo caso, notate la 's' alla fine del nome della funzione) effettua la stessa deserializzazione della funzione `pickle.load()`. Invece di prendere un oggetto stream e leggere i dati serializzati da un file, prende un oggetto bytes contenente dati serializzati, come quello restituito dalla funzione `pickle.dumps()`.
4. Il risultato finale è lo stesso: una riproduzione perfetta del dizionario originale.

*
**

13.5. I BYTE E LE STRINGHE SOLLEVANO ANCORA LA LORO RIPUGNANTE TESTA

Il protocollo pickle esiste da molti anni ed è maturato man mano che lo stesso Python è maturato. Ora ci sono quattro versioni differenti del protocollo pickle.

- Python 1.x aveva due protocolli pickle: un formato basato su testo (“versione 0”) e un formato binario (“versione 1”).
- Python 2.3 ha introdotto un nuovo protocollo pickle (“versione 2”) per gestire nuove funzioni negli oggetti classe di Python. Questo è un formato binario.
- Python 3.0 ha introdotto un altro protocollo pickle (“versione 3”) con supporto esplicito per gli oggetti bytes e gli array di byte. Questo è un formato binario.

Oh, guardate, la differenza tra byte e stringhe solleva la sua ripugnante testa ancora una volta. (Se questo vi sorprende, non siete stati molto attenti.) In pratica, questo significa che mentre Python 3 è in grado di leggere dati serializzati con la versione 2 del protocollo, Python 2 non è in grado di leggere dati serializzati con la versione 3 del protocollo.



13.6. EFFETTUARE IL DEBUG DEI FILE PICKLE

Che aspetto ha il protocollo pickle? Abbandoniamo la Shell Python per un momento e diamo un’occhiata a quel file `entry.pickle` che abbiamo creato.

```

you@localhost:~/diveintopython3/esempi$ ls -l entry.pickle
-rw-r--r-- 1 you  you 365 Aug  3 13:34 entry.pickle
you@localhost:~/diveintopython3/esempi$ cat entry.pickle
comments_linkqNXtagsqXdiveintopythonqXdocbookqXhtmlq?qX publishedq?
Xarticle_linkXJhttp://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition
q  Xpublished_dateq
ctime
struct_time
?qRqXtitleqXImmersione nella storia, edizione 2009qu.

```

Questo non ci aiuta molto. Potete vedere le stringhe, ma gli altri tipi di dato sono composti da caratteri non stampabili (o quantomeno non leggibili). I campi non sono ovviamente delimitati da tabulazioni o spazi. Questo non è un formato di cui vorreste effettuare il debug da soli.

```

>>> shell

1
>>> import pickletools
>>> with open('entry.pickle', 'rb') as f:
...     pickletools.dis(f)
 0: \x80 PROTO      3
 2: }      EMPTY_DICT
 3: q      BININPUT    0
 5: (      MARK
 6: X      BINUNICODE  'published_date'
25: q      BININPUT    1
27: c      GLOBAL      'time struct_time'
45: q      BININPUT    2
47: (      MARK
48: M      BININT2     2009
51: K      BININT1     3
53: K      BININT1     27
55: K      BININT1     22
57: K      BININT1     20
59: K      BININT1     42
61: K      BININT1     4
63: K      BININT1     86
65: J      BININT      -1
70: t      TUPLE      (MARK at 47)
71: q      BININPUT    3
73: }      EMPTY_DICT
74: q      BININPUT    4
76: \x86   TUPLE2
77: q      BININPUT    5
79: R      REDUCE
80: q      BININPUT    6
82: X      BINUNICODE  'comments_link'
100: q     BININPUT    7
102: N     NONE

```

```

103: X      BINUNICODE 'internal_id'
119: q      BINPUT      8
121: C      SHORT_BINBYTES 'pÖ´ø'
127: q      BINPUT      9
129: X      BINUNICODE 'tags'
138: q      BINPUT      10
140: X      BINUNICODE 'diveintopython'
159: q      BINPUT      11
161: X      BINUNICODE 'docbook'
173: q      BINPUT      12
175: X      BINUNICODE 'html'
184: q      BINPUT      13
186: \x87   TUPLE3
187: q      BINPUT      14
189: X      BINUNICODE 'title'
199: q      BINPUT      15
201: X      BINUNICODE 'Immersione nella storia, edizione 2009'
244: q      BINPUT      16
246: X      BINUNICODE 'article_link'
263: q      BINPUT      17
265: X      BINUNICODE 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition
344: q      BINPUT      18
346: X      BINUNICODE 'published'
360: q      BINPUT      19
362: \x88   NEWTRUE
363: u      SETITEMS   (MARK at 5)
364: .      STOP

```

```
highest protocol among opcodes = 3
```

L'informazione più interessante in questo disassemblato si trova sull'ultima riga, perché include la versione del protocollo pickle con la quale questo file è stato salvato. Non c'è alcun contrassegno di versione esplicito nel protocollo pickle. Per determinare la versione del protocollo che è stata usata per memorizzare un file pickle, dovete esaminare i contrassegni (chiamati "opcode" in inglese) all'interno dei dati serializzati e usare la conoscenza cablata di quali opcode sono stati introdotti con ogni versione del protocollo pickle. La funzione

`pickle.dis()` fa proprio questo e stampa il risultato nell'ultima riga del disassemblato. Ecco una funzione che restituisce solo il numero di versione, senza stampare nulla:

```
import pickletools

def protocol_version(file_object):
    maxproto = -1
    for opcode, arg, pos in pickletools.genops(file_object):
        maxproto = max(maxproto, opcode.proto)
    return maxproto
```

Ed eccola qui in azione:

```
>>> import pickleversion
>>> with open('entry.pickle', 'rb') as f:
...     v = pickleversion.protocol_version(f)
>>> v
3
```



13.7. SERIALIZZARE OGGETTI PYTHON PER LEGGERLI CON ALTRI LINGUAGGI

Il formato di dati usato dal modulo `pickle` è specifico per Python. Non cerca in alcun modo di essere compatibile con altri linguaggi di programmazione. Se la compatibilità verso altri linguaggi è uno dei vostri requisiti, dovete rivolgervi a formati di serializzazione differenti. Uno di questi formati è JSON. “JSON” sta per “JavaScript Object Notation” (letteralmente, notazione degli oggetti JavaScript) ma non fatevi ingannare dal nome — JSON è stato esplicitamente progettato per poter essere usato con molteplici linguaggi di programmazione.

Python 3 include un modulo `json` nella libreria standard. Come il modulo `pickle`, anche il modulo `json` è dotato di funzioni per serializzare strutture dati, memorizzare i dati serializzati su disco, caricare i dati serializzati dal disco e deserializzare i dati in un nuovo oggetto Python. Ma ci sono anche alcune importanti differenze. Prima di tutto, il formato dati JSON è basato su testo, non binario. La [RFC 4627](#) definisce il formato JSON e il modo in cui diversi tipi di dato devono essere codificati sotto forma di testo. Per esempio, un valore booleano viene memorizzato come la stringa di cinque caratteri `'false'` oppure come la stringa di quattro caratteri `'true'`. Tutti i valori in JSON sono sensibili alle maiuscole.

Secondo, come con ogni formato basato su testo, c'è il problema degli spazi bianchi. JSON consente la presenza di una quantità arbitraria di spazio bianco (spazi, tabulazioni, ritorni a capo e caratteri di fine riga) tra i valori. Questo spazio bianco è “insignificante”, nel senso che i codificatori JSON possono aggiungere tanto spazio bianco quanto desiderano e i decodificatori JSON sono obbligati a ignorare lo spazio bianco tra i valori. Questo vi permette di “formattare la stampa” dei vostri dati JSON, annidando gradevolmente i valori contenuti in altri valori a differenti livelli di indentazione in modo da poterli leggere in un browser standard o in un editor di testo. Il modulo `json` di Python è dotato di opzioni per formattare i dati stampati durante la codifica.

Terzo, c'è l'eterno problema della codifica di carattere. JSON codifica i valori come testo semplice, ma come sapete il “testo semplice” non esiste. JSON deve essere memorizzato in una codifica Unicode (UTF-32, UTF-16, o la codifica predefinita UTF-8) e la [sezione 3 della RFC 4627](#) definisce il modo in cui distinguere quale codifica è stata usata.



13.8. SALVARE I DATI IN UN FILE JSON

JSON somiglia notevolmente a una struttura dati che potreste definire manualmente in JavaScript. Questa somiglianza non è casuale; potete effettivamente usare la funzione JavaScript `eval()` per “decodificare” i dati serializzati in JSON. (Valgono le solite [avvertenze sugli ingressi non affidabili](#), ma il punto è che JSON è codice JavaScript valido.) In quanto tale, JSON potrebbe già sembrarvi familiare.

```

>>> shell
1
>>> basic_entry = {} ①
>>> basic_entry['id'] = 256
>>> basic_entry['title'] = 'Immersione nella storia, edizione 2009'
>>> basic_entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> basic_entry['published'] = True
>>> basic_entry['comments_link'] = None
>>> import json
>>> with open('basic.json', mode='w', encoding='utf-8') as f: ②
...     json.dump(basic_entry, f) ③

```

1. Creeremo una nuova struttura dati invece di riutilizzare la struttura dati entry esistente. Più avanti in questo capitolo vedremo cosa succede quando proviamo a codificare strutture dati più complesse in JSON.
2. JSON è un formato basato su testo e ciò significa che dovete aprire questo file in modalità testo e specificare una codifica di carattere. Non potete sbagliare se scegliete UTF-8.
3. Come il modulo pickle, il modulo json definisce una funzione dump() che prende una struttura dati Python e un oggetto stream aperto in scrittura. La funzione dump() serializza la struttura dati Python e la scrive sull'oggetto stream. Effettuare questa operazione all'interno di un'istruzione with ci garantisce che il file verrà opportunamente chiuso quando avremo finito di lavorare.

Quindi che aspetto ha la serializzazione JSON risultante?

```

you@localhost:~/diveintopython3/esempi$ cat basic.json
{"published": true, "tags": ["diveintopython", "docbook", "html"], "comments_link": null,
"id": 256, "title": "Immersione nella storia, edizione 2009"}

```

Questo è certamente più leggibile di un file pickle. Ma JSON può contenere una quantità arbitraria di spazio bianco tra i valori, e il modulo json fornisce un modo facile per beneficiare di questa caratteristica creando file JSON ancora più leggibili.

```
>>> shell
1
>>> with open('basic-pretty.json', mode='w', encoding='utf-8') as f:
...     json.dump(basic_entry, f, indent=2) ①
```

- I. Se passate un parametro `indent` alla funzione `json.dump()`, essa renderà il file JSON risultante più leggibile a spese di una maggiore dimensione del file. Il parametro `indent` è un intero: 0 significa “metti ogni valore su una riga separata”; un numero più grande di 0 significa “metti ogni valore su una riga separata e usa questo numero di spazi per indentare le strutture dati annidate”.

E questo è il risultato:

```
you@localhost:~/diveintopython3/esempi$ cat basic-pretty.json
{
    "published": true,
    "tags": [
        "diveintopython",
        "docbook",
        "html"
    ],
    "comments_link": null,
    "id": 256,
    "title": "Immersione nella storia, edizione 2009"
}
```

*
**

13.9. CORRELARE I TIPI DI DATO PYTHON A JSON

Dato che il formato JSON non è specifico per Python, ci sono alcuni sfasamenti nella sua copertura dei tipi di dato Python. Alcuni di questi sono semplicemente differenze di nome, ma ci sono due importanti tipi di dato Python che sono completamente assenti. Vedete se riuscite a capire quali sono.

Note	JSON	Python 3
	oggetto	dizionario
	array	lista
	stringa	stringa
	intero	intero
	numero reale	numero in virgola mobile
*	true	True
*	false	False
*	null	None
* Tutti i valori JSON sono sensibili alle maiuscole.		

Avete notato cosa manca? Tuple & byte! JSON ha un tipo array, che il modulo json mette in correlazione con una lista Python, ma non ha un tipo separato per gli “array congelati” (cioè tuple). E mentre JSON supporta le stringhe piuttosto bene, non ha alcun supporto per gli oggetti bytes o gli array di byte.



13.10. SERIALIZZARE TIPI DI DATO NON SUPPORTATI DA JSON

Anche se JSON non ha un supporto built-in per i byte, questo non significa che non potete serializzare gli oggetti bytes. Il modulo json fornisce alcuni agganci estendibili per codificare e decodificare tipi di dato sconosciuti. (Per “sconosciuti” intendo “non definiti in JSON”. Ovviamente il modulo json conosce gli array di byte, ma è vincolato dalle limitazioni della specifica JSON.) Se volete codificare byte o altri tipi di dato che JSON non supporta nativamente, dovete fornire codificatori e decodificatori personalizzati per quei tipi.

```

>>> shell
1
>>> entry ①
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Immersione nella storia, edizione 2009',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20, tm_sec=0),
 'published': True}
>>> import json
>>> with open('entry.json', 'w', encoding='utf-8') as f: ②
...     json.dump(entry, f) ③
...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "C:\Python31\lib\json\encoder.py", line 170, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: b'\xDE\xD5\xB4\xF8' is not JSON serializable

```

1. Bene, è il momento di rivisitare la struttura dati entry. Ha tutto: un valore booleano, un valore None, una tupla di stringhe, un oggetto bytes e una struttura temporale dal modulo time.
2. So di averlo già detto, ma vale la pena ripeterlo: JSON è un formato basato su testo. Aprite sempre i file JSON in modalità testo con una codifica di carattere UTF-8.
3. Be', questo non va bene. Cos'è successo?

Ecco cos'è successo: la funzione `json.dump()` ha tentato di serializzare l'oggetto bytes `b'\xDE\xD5\xB4\xF8'`, ma ha fallito perché JSON non è dotato di alcun supporto per gli oggetti bytes. Tuttavia, se per voi memorizzare i byte è importante, potete definire il vostro “mini-formato di serializzazione”.

```
def to_json(python_object):                                ①
    if isinstance(python_object, bytes):                  ②
        return {'__class__': 'bytes',
                '__value__': list(python_object)}          ③
    raise TypeError(repr(python_object) + ' non è serializzabile in JSON') ④
```

1. Per definire il vostro “mini-formato di serializzazione” per un tipo di dato che JSON non supporta nativamente, vi basta definire una funzione che prende un oggetto Python come parametro. Questo oggetto Python sarà l'effettivo oggetto che la funzione `json.dump()` non è in grado di serializzare da sola — in questo caso, l'oggetto `b'\xDE\xD5\xB4\xF8'` di tipo bytes.
2. La vostra funzione di serializzazione personalizzata dovrebbe controllare il tipo dell'oggetto Python che la funzione `json.dump()` le ha passato. Questo non è strettamente necessario se la vostra funzione serializza un solo tipo di dato, ma chiarisce qual è il caso che la vostra funzione sta affrontando e facilita l'estensione se più tardi avete bisogno di aggiungere serializzazioni per altri tipi di dato.
3. In questo caso, ho scelto di convertire un oggetto bytes in un dizionario. La chiave `__class__` manterrà il tipo di dato originale (sotto forma di stringa, 'bytes') e la chiave `__value__` manterrà l'effettivo valore. Naturalmente, questo non può essere un oggetto bytes; il punto è proprio quello di convertire l'oggetto in qualcosa che possa essere serializzato in JSON! Un oggetto bytes è solo una sequenza di interi, ognuno compreso nell'intervallo 0–255. Possiamo usare la funzione `list()` per convertire l'oggetto bytes in una lista di interi, in modo che `b'\xDE\xD5\xB4\xF8'` diventi `[222, 213, 180, 248]`. (Fate i calcoli! Funziona! Il byte `\xDE` in esadecimale corrisponde a 222 in decimale, `\xD5` corrisponde a 213, e così via.)
4. Questa riga è importante. La struttura dati che state serializzando potrebbe contenere tipi che né il serializzatore built-in JSON né il vostro serializzatore personalizzato sono in grado di maneggiare. In questo caso, il vostro serializzatore personalizzato deve sollevare un'eccezione di tipo `TypeError` in modo che la funzione `json.dump()` sappia che il vostro serializzatore personalizzato non ha riconosciuto il tipo di dato.

Questo è tutto, non dovete fare altro. In particolare, questa funzione di serializzazione personalizzata *restituisce un dizionario Python*, non una stringa. Non state completando l'intera serializzazione in JSON da soli, ma state solamente effettuando la conversione in un tipo di dato supportato. La funzione `json.dump()` farà il resto.

```
>>> shell
```

```
1
```

```
>>> import customserializer
```

①

```
>>> with open('entry.json', 'w', encoding='utf-8') as f:
```

②

```
...     json.dump(entry, f, default=customserializer.to_json)
```

③

```
...
```

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

```
File "<stdin>", line 9, in <module>
```

```
    json.dump(entry, f, default=customserializer.to_json)
```

```
File "C:\Python31\lib\json\__init__.py", line 178, in dump
```

```
    for chunk in iterable:
```

```
File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
```

```
    for chunk in _iterencode_dict(o, _current_indent_level):
```

```
File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
```

```
    for chunk in chunks:
```

```
File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
```

```
    o = _default(o)
```

```
File "/Users/pilgrim/diveintopython3/esempi/customserializer.py", line 12, in to_json
```

```
    raise TypeError(repr(python_object) + ' non è serializzabile in JSON')
```

④

```
TypeError: time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20, tm_sec=42, tm_w
```

1. Il modulo `customserializer` è dove avete appena definito la funzione `to_json()` dell'esempio precedente.
2. Modalità testo, codifica UTF-8, bla bla bla. (Lo dimenticherete! Me lo dimentico anch'io, ogni tanto! E tutto funzionerà esattamente fino al momento in cui smetterà di funzionare, e in quel momento tutto si bloccherà in maniera spettacolare.)
3. Questa è la parte importante: per agganciare la vostra funzione di conversione personalizzata alla funzione `json.dump()`, passate la vostra funzione alla funzione `json.dump()` nel parametro `default`. (Urrà, ogni cosa in Python è un oggetto!)
4. Bene, e quindi in realtà non ha funzionato. Ma date un'occhiata all'eccezione. La funzione `json.dump()` non si sta più lamentando di non riuscire a serializzare l'oggetto `bytes`. Ora si lamenta di un oggetto completamente differente: l'oggetto `time.struct_time`.

Sebbene ottenere un'eccezione differente possa non sembrare un passo avanti, in realtà lo è! Ci vorrà solo un altro ritocco per risolvere il problema.

```

import time

def to_json(python_object):
    if isinstance(python_object, time.struct_time):           ①
        return {'__class__': 'time.asctime',
                '__value__': time.asctime(python_object)}      ②
    if isinstance(python_object, bytes):
        return {'__class__': 'bytes',
                '__value__': list(python_object)}
    raise TypeError(repr(python_object) + ' non è serializzabile in JSON')

```

1. Aggiungendo codice alla nostra funzione `customserializer.to_json()` esistente, dobbiamo controllare se l'oggetto Python (con cui la funzione `json.dump()` ha dei problemi) è un'istanza della classe `time.struct_time`.
2. Se è così, faremo qualcosa di simile alla conversione che abbiamo compiuto per l'oggetto `bytes`: convertiamo l'oggetto `time.struct_time` in un dizionario che contiene solo valori serializzabili in formato JSON. In questo caso, il modo più semplice di convertire una data in un valore serializzabile in formato JSON è quello di convertirla in una stringa tramite la funzione `time.asctime()`. La funzione `time.asctime()` convertirà lo sconveniente oggetto di tipo `time.struct_time` nella stringa `'Fri Mar 27 22:20:42 2009'`.

Con queste due conversioni personalizzate, l'intera struttura dati `entry` dovrebbe venire serializzata in JSON senza ulteriori problemi.

```

>>> shell
1
>>> with open('entry.json', 'w', encoding='utf-8') as f:
...     json.dump(entry, f, default=customserializer.to_json)
...

```



```
you@localhost:~/diveintopython3/esempi$ ls -l example.json
-rw-r--r-- 1 you  you  398 Aug  3 13:34 entry.json
you@localhost:~/diveintopython3/esempi$ cat example.json
{"published_date": {"__class__": "time.asctime", "__value__": "Fri Mar 27 22:20:42 2009"},
"comments_link": null, "internal_id": {"__class__": "bytes", "__value__": [222, 213, 180, 248]},
"tags": ["diveintopython", "docbook", "html"], "title": "Immersione nella storia, edizione 2009",
"article_link": "http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition",
"published": true}
```



13.11. CARICARE DATI DA UN FILE JSON

Come il modulo `pickle`, anche il modulo `json` è dotato di una funzione `load()` che prende un oggetto stream, legge dati codificati in JSON dal flusso e crea un nuovo oggetto Python che rispecchia la struttura dati JSON.

```

>>> shell
2
>>> del entry ①
>>> entry
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import json
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry = json.load(f) ②
...
>>> entry ③
{'comments_link': None,
 'internal_id': {'__class__': 'bytes', '__value__': [222, 213, 180, 248]},
 'title': 'Immersione nella storia, edizione 2009',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': {'__class__': 'time.asctime', '__value__': 'Fri Mar 27 22:20:42 2009'},
 'published': True}

```

1. A scopo dimostrativo, spostatevi nella Shell Python n°2 e cancellate la struttura dati entry che avete creato precedentemente in questo capitolo utilizzando il modulo `pickle`.
2. Nel caso più semplice, la funzione `json.load()` opera allo stesso modo della funzione `pickle.load()`. Passate un oggetto stream e vi viene restituito un nuovo oggetto Python.
3. Ho una buona e una cattiva notizia. Prima la buona notizia: la funzione `json.load()` legge con successo il file `entry.json` che avete creato nella Shell Python n°1 e crea un nuovo oggetto Python che contiene i dati. Ora la cattiva notizia: la funzione non ha ricreato la struttura dati entry originale. I due valori `'internal_id'` e `'published_date'` sono stati ricreati sotto forma di dizionari — nello specifico, dizionari contenenti i valori compatibili con JSON che avete creato nella funzione di conversione `to_json()`.

La funzione `json.load()` non sa nulla di qualunque funzione di conversione possiate aver passato a `json_dump()`. Ciò di cui avete bisogno è l'opposto della funzione `to_json()` — una funzione che prenda un oggetto JSON proveniente da una conversione personalizzata e lo converta all'indietro nel tipo di dato Python originale.

```
# aggiungete questa funzione a customserializer.py

def from_json(json_object): ①
    if '__class__' in json_object: ②
        if json_object['__class__'] == 'time.asctime':
            return time.strptime(json_object['__value__']) ③
        if json_object['__class__'] == 'bytes':
            return bytes(json_object['__value__']) ④
    return json_object
```

1. Anche questa funzione di conversione prende un parametro e restituisce un valore. Ma il parametro che accetta non è una stringa, bensì un oggetto Python — il risultato della deserializzazione in Python di una stringa codificata in JSON.
2. Tutto quello che vi serve è controllare se questo oggetto contiene la chiave '`__class__`' che la funzione `to_json()` aveva creato. Se è così, il valore della chiave '`__class__`' vi dirà come decodificare il valore nel suo tipo di dato Python originale.
3. Per decodificare la stringa restituita dalla funzione `time.asctime()` che contiene la data, usate la funzione `time.strptime()`. Questa funzione prende una stringa che contiene una data formattata (in un formato personalizzabile, ma usa lo stesso formato predefinito della funzione `time.asctime()`) e restituisce un oggetto `time.struct_time`.
4. Per convertire una lista di interi in un oggetto `bytes`, potete usare la funzione `bytes()`.

E questo è tutto: c'erano solo due tipi di dato gestiti nella funzione `to_json()` e ora quei due tipi di dato sono gestiti anche nella funzione `from_json()`. Questo è il risultato:

```

>>> shell
2
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry = json.load(f, object_hook=customserializer.from_json) ①
...
>>> entry ②
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Immersione nella storia, edizione 2009',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20, tm_sec=0),
 'published': True}

```

1. Per agganciare la funzione `from_json()` al processo di deserializzazione, passatela come parametro `object_hook` alla funzione `json.load()`. Funzioni che accettano funzioni come parametri: è così comodo!
2. Ora la struttura dati `entry` contiene una chiave `'internal_id'` il cui valore è un oggetto `bytes`. Contiene anche una chiave `'published_date'` il cui valore è un oggetto `time.struct_time`.

C'è ancora un ultimo difetto, però.

```

>>> shell
1
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry2 = json.load(f, object_hook=customserializer.from_json)
...
>>> entry2 == entry ①
False
>>> entry['tags'] ②
('diveintopython', 'docbook', 'html')
>>> entry2['tags'] ③
['diveintopython', 'docbook', 'html']

```

1. Anche dopo aver agganciato la funzione `to_json()` al processo di serializzazione e aver agganciato la funzione `from_json()` a quello di deserializzazione, non abbiamo ancora ricreato una riproduzione perfetta della struttura dati originale. Come mai?
2. Nella struttura dati `entry` originale, il valore della chiave `'tags'` era una tupla di tre stringhe.
3. Ma nella struttura dati `entry2` il valore della chiave `'tags'` è diventato una *lista* di tre stringhe a seguito del viaggio di andata e ritorno tra serializzazione e deserializzazione. JSON non distingue le tuple dalle liste; usa gli array come singolo tipo di dato simile alle liste, così il modulo `json` converte silenziosamente sia le tuple che le liste in array JSON durante la serializzazione. La maggior parte delle volte potete ignorare la differenza tra tuple e liste, ma questo sfasamento è una cosa da tenere presente quando lavorate con il modulo `json`.

13.12. LETTURE DI APPROFONDIMENTO



Molti articoli sul modulo `pickle` fanno riferimento a `cPickle`. In Python 2, c'erano due implementazioni del modulo `pickle`, una scritta in Python e una scritta in C (ma invocabile da Python). In Python 3, questi due moduli sono stati unificati, quindi dovrete sempre scrivere semplicemente `import pickle`. Potreste trovare utili questi articoli, ma dovrete ignorare il riferimento ormai obsoleto a `cPickle`.

Sulla serializzazione tramite il modulo `pickle`:

- Il modulo `pickle`
- `pickle` e `cPickle` — la serializzazione di oggetti Python
- Usare `pickle`
- La gestione della persistenza in Python

Su JSON e il modulo `json`:

- `json` — il serializzatore in JavaScript Object Notation
- La codifica e la decodifica JSON con oggetti personalizzati in Python

Sulla estendibilità del protocollo `pickle`:

- Serializzare istanze di classi
- La persistenza di oggetti esterni
- Gestire oggetti con stato

CAPITOLO 14. SERVIZI WEB HTTP

“ Non c’è cuscino più morbido di una coscienza tranquilla. ”

— Charlotte Brontë

14.1. IMMERSIONE!

Relativamente alla descrizione della natura dei servizi web HTTP, non occorrono più di 15 parole: lo scambio di dati con server remoti utilizzando nient’altro che le operazioni di HTTP. Se volete ottenere dati dal server, usate HTTP GET; se volete inviare nuovi dati al server, usate HTTP POST. Alcune delle API più avanzate per i servizi web HTTP permettono anche di creare, modificare e cancellare dati utilizzando HTTP PUT e HTTP DELETE. In altre parole, i “verbi” definiti dal protocollo HTTP (GET, POST, PUT e DELETE) possono corrispondere direttamente a operazioni a livello di applicazione per recuperare, creare, modificare e cancellare dati.

Il vantaggio principale di questo approccio è la semplicità, e la sua semplicità si è dimostrata popolare. I dati — tipicamente in formato XML o JSON — possono essere assemblati e memorizzati staticamente oppure generati dinamicamente da un programma lato server, e tutti i linguaggi di programmazione più importanti (compreso Python, naturalmente!) includono una libreria HTTP per scaricarli. Anche le attività di debug vengono facilitate: dato che ogni risorsa in un servizio web HTTP ha un indirizzo unico (sotto forma di URL), potete caricarla nel vostro browser web e vederne immediatamente i dati grezzi.

Esempi di servizi web HTTP:

- le API di Google Data vi permettono di interagire con un’ampia varietà di servizi forniti da Google, compresi Blogger e YouTube;
- i Servizi Flickr vi permettono di caricare e scaricare foto da Flickr;
- la API di Twitter vi permette di pubblicare aggiornamenti di stato su Twitter;
- ...e molti altri.

Python 3 viene distribuito con due diverse librerie per interagire con i servizi web HTTP:

- `http.client` è una libreria di basso livello che implementa la RFC 2616, cioè il protocollo HTTP;
- `urllib.request` è un livello di astrazione costruito sulla base di `http.client`. Fornisce una API standard per accedere sia ai server HTTP sia ai server FTP, segue automaticamente le redirezioni HTTP e gestisce alcune forme comuni di autenticazione HTTP.

Quindi quale delle due dovrete usare? Nessuna. Invece, dovrete usare `httplib2`, una libreria open source di terze parti che implementa HTTP in maniera più completa rispetto ad `http.client` ma fornisce astrazioni migliori rispetto a `urllib.request`.

Per comprendere perché `httplib2` è la scelta giusta dovete prima conoscere HTTP.



14.2. LE CARATTERISTICHE DI HTTP

Ci sono cinque importanti caratteristiche che tutti i client HTTP dovrebbero supportare.

14.2.1. LA CACHE

La cosa più importante da capire per un qualsiasi tipo di servizio web è che l'accesso alla rete è incredibilmente costoso. Non nel senso di “euro e centesimi” (sebbene la banda non sia gratis). Voglio dire che ci vuole un tempo estremamente lungo per aprire una connessione, spedire una richiesta e ricevere una risposta da un server remoto. Anche sulla connessione a banda larga più veloce, la *latenza* (il tempo che ci vuole per spedire una richiesta e cominciare a ricevere i dati in una risposta) può ancora essere più grande di quanto prevedete. Un router si comporta in modo strano, un pacchetto viene scartato, un proxy intermedio è sotto attacco — non c'è mai un momento di calma sulla rete Internet pubblica, e potrebbe non esserci nulla che voi possiate fare.

HTTP è progettato con l'uso della cache in mente. Esiste un'intera categoria di dispositivi (chiamati "proxy di cache") il cui solo lavoro è sedere in mezzo tra voi e il resto del mondo e minimizzare l'accesso alla rete. La vostra azienda o il vostro ISP quasi certamente gestisce alcuni proxy di cache, anche se voi non ve ne rendete conto. Questi dispositivi funzionano perché l'uso della cache è integrato nel protocollo HTTP.

Ecco un esempio concreto di come funziona la cache. Visitate diveintomark.org nel vostro browser. Quella pagina include un'immagine di sfondo, wearehugh.com/m.jpg. Quando il vostro browser scarica quell'immagine, il server include le seguenti intestazioni HTTP:

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
```

```
Cache-Control: max-age=31536000, publicExpires: Mon, 31 May 2010 17:14:04 GMT
```

```
Connection: close
```

```
Content-Type: image/jpeg
```

Le intestazioni Cache-Control ed Expires dicono al vostro browser (e a qualsiasi proxy di cache tra voi e il server) che questa immagine può essere tenuta in cache per un anno. *Un anno!* E se nel prossimo anno visitate un'altra pagina che include un collegamento a questa immagine, il vostro browser caricherà l'immagine dalla propria cache senza generare alcun tipo di attività di rete.

Cache-
Control:
max-age
significa:
"Non
scocciarmi
fino alla
prossima
settimana."

Ma aspettate, le cose migliorano. Diciamo che per qualche motivo il vostro browser cancella l'immagine dalla vostra cache locale. Magari ha finito lo spazio su disco, magari voi avete pulito la cache manualmente. Ma le intestazioni HTTP dicevano che questi dati potevano essere memorizzati in cache da proxy di cache pubblici. (Tecnicamente, la cosa importante è quello che le intestazioni *non* dicono: l'intestazione `Cache-Control` non contiene la parola chiave `private`, quindi questi dati sono memorizzabili in cache per default.) I proxy di cache sono progettati per avere tonnellate di spazio di memorizzazione, probabilmente molto più di quanto abbiate allocato per il vostro browser locale.

Se la vostra azienda o il vostro ISP gestisce un proxy di cache, il proxy potrebbe ancora avere l'immagine nella propria cache. Quando visitate `diveintomark.org` un'altra volta, il vostro browser cercherà l'immagine nella sua cache locale ma non la troverà, quindi effettuerà una richiesta di rete per cercare di scaricarla dal server remoto. Ma se il proxy di cache ha ancora una copia dell'immagine, intercetterà quella richiesta e preleverà l'immagine dalla *propria* cache. Questo significa che la vostra richiesta non raggiungerà mai il server remoto; in effetti, non lascerà mai la rete della vostra azienda. Questo rende lo scaricamento più veloce (meno salti sui nodi della rete) e fa risparmiare denaro alla vostra azienda (meno dati che vengono scaricati dal mondo esterno).

L'uso della cache in HTTP funziona solo quando tutti fanno la loro parte. Da un lato, i server devono spedire le intestazioni corrette nelle loro risposte. Dall'altro lato, i client devono comprendere e rispettare quelle intestazioni prima di richiedere due volte gli stessi dati. I proxy nel mezzo non sono una panacea; possono essere tanto intelligenti solo quanto i server e i client permettono loro di essere.

Le librerie HTTP incluse in Python non supportano l'uso della cache, ma `httplib2` lo fa.

14.2.2. IL CONTROLLO DELLA MODIFICA PIÙ RECENTE

Alcuni dati non cambiano mai, mentre altri dati cambiano continuamente. Nel mezzo, c'è un vasto assortimento di dati che *potrebbero* essere cambiati ma non lo hanno fatto. Il feed di CNN.com viene aggiornato ogni pochi minuti, ma il feed del mio weblog potrebbe non cambiare per diversi giorni o intere settimane. In quest'ultimo caso, non voglio dire ai client di mantenere in cache il mio feed per intere settimane, perché quando scrivo effettivamente qualcosa i miei lettori potrebbero non vederla per settimane (perché stanno rispettando le mie intestazioni di cache che dicono “non preoccuparti di controllare questo feed per settimane”). D'altra parte, non voglio che i client scarichino il mio intero feed una volta ogni ora se non è cambiato!

HTTP fornisce una soluzione anche a questo problema. Quando richiedete un dato per la prima volta, il server può spedire indietro un'intestazione Last-Modified. Questa è esattamente ciò che sembra: la data in cui i dati sono stati modificati. Quell'immagine di sfondo riferita da diveintomark.org includeva un'intestazione Last-Modified.

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

304 Not
Modified
significa:
*“Stessa
merda, altro
giorno.”*

Quando richiedete gli stessi dati una seconda (o terza, o quarta) volta, potete spedire insieme alla vostra richiesta un'intestazione If-Modified-Since contenente la data che avete ottenuto dal server l'ultima volta. Se i dati sono cambiati da allora, il server ignora l'intestazione If-Modified-Since e vi restituisce semplicemente i nuovi dati con un codice di stato 200. Ma se i dati *non* sono cambiati, il server rimanda indietro lo speciale codice di stato HTTP 304 per indicare che “questi dati non sono cambiati dall'ultima volta che li avete richiesti”. Potete verificare questo comportamento dalla riga di comando usando `curl`:

```
you@localhost:~$ curl -I -H "If-Modified-Since: Fri, 22 Aug 2008 04:28:16 GMT" http://wearehugh.com/m.  
HTTP/1.1 304 Not Modified  
Date: Sun, 31 May 2009 18:04:39 GMT  
Server: Apache  
Connection: close  
ETag: "3075-ddc8d800"  
Expires: Mon, 31 May 2010 18:04:39 GMT  
Cache-Control: max-age=31536000, public
```

Perché questo è un miglioramento? Perché quando il server restituisce un 304, *non rispedisce i dati*. Tutto quello che ottenete è il codice di stato. Anche dopo che la copia nella vostra cache è scaduta, il controllo della modifica più recente vi assicura che non scaricherete gli stessi dati due volte se non sono cambiati. (Come bonus aggiuntivo, questa risposta 304 include anche le intestazioni di cache. I proxy manterranno una copia dei dati anche dopo che sono ufficialmente “scaduti”, nella speranza che i dati non siano *realmente* cambiati e che la prossima richiesta ottenga una risposta con un codice di stato 304 e informazioni di cache aggiornate.)

Le librerie HTTP incluse in Python non supportano il controllo della modifica più recente, ma `httplib2` lo fa.

14.2.3. IL CONTROLLO DEGLI ETAG

Gli ETag sono un modo alternativo per effettuare il controllo della modifica più recente. Con gli ETag, il server spedisce un codice hash in un'intestazione ETag insieme ai dati che avete richiesto. (Decidere come determinare esattamente questo hash è interamente compito del server. L'unico requisito è che l'hash cambi quando i dati cambiano.) Quell'immagine di sfondo riferita da `diveintomark.org` aveva un'intestazione ETag.

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

La seconda volta che richiedete gli stessi dati, includete il valore di ETag in un'intestazione If-None-Match della vostra richiesta. Se i dati non sono cambiati, il server vi restituirà un codice di stato 304. Come con il controllo sulla data della modifica più recente, il server invia *solo* il codice di stato 304, evitando di spedirvi gli stessi dati una seconda volta. Includendo il valore di ETag nella vostra seconda richiesta state dicendo al server che non c'è alcun bisogno di rispedire gli stessi dati se quei dati corrispondono ancora a questo hash, dato che avete ancora i dati dall'ultima volta.

Utilizzando ancora curl:

```
you@localhost:~$ curl -I -H "If-None-Match: \"3075-ddc8d800\"" http://wearehugh.com/m.jpg ①
HTTP/1.1 304 Not Modified
Date: Sun, 31 May 2009 18:04:39 GMT
Server: Apache
Connection: close
ETag: "3075-ddc8d800"
Expires: Mon, 31 May 2010 18:04:39 GMT
Cache-Control: max-age=31536000, public
```

ETag
significa:
*“Nulla di
nuovo sotto il
sole.”*


- I. Gli ETag vengono comunemente racchiusi tra virgolette, ma *le virgolette sono parte del valore*. Questo significa che dovete rimandare le virgolette al server nell'intestazione `If-None-Match`.

Le librerie HTTP incluse in Python non supportano gli ETag, ma `httplib2` lo fa.

14.2.4. LA COMPRESSIONE

Quando parlate di servizi web HTTP, state quasi certamente parlando di spostare dati basati su testo avanti e indietro attraverso la rete. Forse i dati sono in formato XML, forse sono in formato JSON, forse sono solo testo semplice. A prescindere dal formato, il testo si comprime bene. Il feed di esempio nel capitolo su XML è di 3070 byte, ma sarebbe di 941 byte dopo una compressione effettuata tramite gzip, esattamente il 30% della dimensione originale!

HTTP supporta diversi algoritmi di compressione. I due tipi più comuni sono gzip e deflate. Quando richiedete una risorsa via HTTP, potete chiedere al server di mandarvela in formato compresso includendo nella vostra richiesta un'intestazione `Accept-encoding` che elenchi quali algoritmi di compressione supportate. Se il server supporta uno qualsiasi degli stessi algoritmi, vi risponderà inviando i dati compressi (insieme a un'intestazione `Content-encoding` che vi dice quale algoritmo ha usato). Poi sta a voi decomprimere i dati.

 Suggerimento importante per gli sviluppatori lato server: assicuratevi che la versione compressa di una risorsa abbia un Etag differente rispetto alla versione non compressa. Altrimenti, i proxy di cache si confonderanno e potrebbero servire la versione compressa a client che non sono in grado di gestirla. Leggete la discussione sul bug 39727 di Apache per maggiori dettagli su questo sottile problema.

Le librerie HTTP incluse in Python non supportano la compressione, ma `httplib2` lo fa.

14.2.5. LE REDIREZIONI

Gli URI fighi non cambiano, ma molti URI sono seriamente sfigati. I siti web vengono riorganizzati, le pagine vengono spostate a nuovi indirizzi, persino i servizi web possono venire riorganizzati. Un feed pubblicato all'indirizzo `http://example.com/index.xml` potrebbe venire spostato all'indirizzo `http://example.com/xml/`

atom.xml. Oppure un intero dominio potrebbe spostarsi, man mano che un'organizzazione si espande e viene ristrutturata: `http://www.example.com/index.xml` diventa `http://server-farm-1.example.com/index.xml`.

Ogni volta che richiedete un qualsiasi tipo di risorsa da un server HTTP, il server include un codice di stato nella propria risposta. Il codice di stato 200 significa “è tutto normale, ecco la pagina che avete chiesto”. Il codice di stato 404 significa “pagina non trovata”. (Avete probabilmente visto errori 404 navigando sul web.) I codici di stato che cominciano con 3 indicano una qualche forma di redirectione.

HTTP ha molti modi diversi per dire che una risorsa si è spostata. Le due tecniche più comuni sono i codici di stato 302 e 301. Il codice di stato 302 indica una *redirezione temporanea*: significa “oops, quella risorsa è stata temporaneamente spostata qui” (e poi vi dà l'indirizzo temporaneo in un'intestazione Location). Il codice di stato 301 indica una *redirezione permanente*: significa “oops, quella risorsa è stata permanentemente spostata” (e poi vi dà il nuovo indirizzo in un'intestazione Location). Se ottenete un codice di stato 302 e un nuovo indirizzo, la specifica HTTP dice che dovrete usare il nuovo indirizzo per ottenere quello che avete chiesto, ma la prossima volta che volete accedere alla stessa risorsa dovrete riprovare il vecchio indirizzo. Se invece ottenete un codice di stato 301 e un nuovo indirizzo, siete tenuti a utilizzare il nuovo indirizzo da quel momento in poi.

Il modulo `urllib.request` “segue” automaticamente le redirectioni quando riceve i codici di stato appropriati da un server HTTP, ma non vi dice di averlo fatto. Finirete per ottenere i dati che avete chiesto, ma non saprete mai che la libreria sottostante vi ha “aiutato” a seguire una redirectione. Così continuerete a tempestare di richieste il vecchio indirizzo, e ogni volta il modulo `urllib.request` vi “aiuterà” a seguire la redirectione. In altre parole, il modulo tratta le redirectioni permanenti allo stesso modo delle redirectioni temporanee. Questo significa due viaggi invece di uno, che è male per il server è male per voi.

`httplib2` gestisce le redirectioni permanenti per voi. Non solo vi dirà che c'è stata una redirectione permanente, ma terrà traccia di tali redirectioni localmente e riscriverà automaticamente gli URL rediretti prima di spedire loro una richiesta.

Location
significa:
“Guarda là!”



14.3. COME NON PRELEVARE DATI VIA HTTP

Diciamo che volete scaricare una risorsa via HTTP, per esempio un feed Atom. Essendo un feed, non vi limiterete a scaricare la risorsa una sola volta, ma vorrete scaricarla più e più volte. (La maggior parte dei lettori di feed controllano ogni ora se ci sono stati dei cambiamenti.) Facciamolo prima in modo veloce ma grossolano, poi vediamo come potete farlo meglio.

```
>>> import urllib.request
>>> a_url = 'http://diveintopython3.org/examples/feed.xml'
>>> data = urllib.request.urlopen(a_url).read() ①
>>> type(data) ②
<class 'bytes'>
>>> print(data)
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='it'>
  <title>dive into mark</title>
  <subtitle>attualmente tra una dipendenza e l'altra</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
  ...
```

1. Scaricare qualsiasi cosa via HTTP è incredibilmente facile in Python; in effetti, vi ci vuole solo una riga di codice. Il modulo `urllib.request` ha una comoda funzione `urlopen()` che prende l'indirizzo della pagina che volete e restituisce un oggetto simile a un file che potete semplicemente leggere con il metodo `read()` per ottenere il contenuto completo della pagina. Non potrebbe proprio essere più facile.
2. Il metodo `urlopen().read()` restituisce sempre un oggetto bytes, non una stringa. Ricordatevi che i byte sono byte e che i caratteri sono un'astrazione. I server HTTP non si occupano di astrazioni. Se richiedete una risorsa, ottenete byte. Se volete una stringa, dovrete determinare la codifica di carattere dei byte ed effettuare esplicitamente la conversione.

Quindi cosa c'è di sbagliato in questo modo di operare? Per un impiego rapido e occasionale durante il collaudo o lo sviluppo, non c'è niente di sbagliato. Lo faccio sempre. Volevo i contenuti del feed e ho ottenuto i contenuti del feed. La stessa tecnica funziona per tutte le pagine web. Ma una volta che cominciate a pensare in termini di un servizio web che volete accedere regolarmente (e.g. richiedendo questo feed una volta ogni ora), allora lo state facendo in maniera inefficiente e grossolana.



14.4. COSA VIENE TRASMESSO ATTRAVERSO LA RETE?

Per vedere perché questo modo di prelevare dati è inefficiente e grossolano, attiviamo le caratteristiche di debug della libreria HTTP inclusa in Python e vediamo cosa viene trasmesso “sul filo” (cioè attraverso la rete).

```
>>> from http.client import HTTPConnection
>>> HTTPConnection.debuglevel = 1 ①
>>> from urllib.request import urlopen
>>> response = urlopen('http://diveintopython3.org/examples/feed.xml') ②
send: b'GET /examples/feed.xml HTTP/1.1 ③
Host: diveintopython3.org ④
Accept-Encoding: identity ⑤
User-Agent: Python-urllib/3.1' ⑥
Connection: close
reply: 'HTTP/1.1 200 OK'
...ulteriori informazioni di debug omesse...
```

- I. Come avevo menzionato all’inizio del capitolo, `urllib.request` si basa su un'altra libreria standard inclusa in Python, `http.client`. Di solito non avete bisogno di toccare `http.client` direttamente. (Il modulo `urllib.request` la importa automaticamente.) Ma qui noi la importiamo in modo da poter attivare il flag di debug sulla classe `HTTPConnection` che `urllib.request` usa per connettersi a un server HTTP.

2. Ora che il flag di debug è impostato, le informazioni sulla richiesta e sulla risposta HTTP sono stampate in tempo reale. Come potete vedere, quando richiedete il feed Atom il modulo `urllib.request` invia cinque righe al server.
3. La prima riga specifica il verbo HTTP che state usando e il percorso della risorsa (senza il nome del dominio).
4. La seconda riga specifica il nome del dominio da dove stiamo richiedendo questo feed.
5. La terza riga specifica gli algoritmi di compressione che il client supporta. Come avevo menzionato in precedenza, `urllib.request` non supporta la compressione di default.
6. La quarta riga specifica il nome della libreria che sta effettuando la richiesta. Per default, questo nome è `Python-urllib` più un numero di versione. Sia `urllib.request` che `httplib2` permettono di cambiare il nome dell'applicazione usata semplicemente aggiungendo un'intestazione `User-Agent` alla richiesta (che sostituirà il valore predefinito).

Ora diamo un'occhiata ai dati che il server ha inviato nella sua risposta.

*Stiamo
scaricando
3070 byte
mentre ne
avremmo
potuti
scaricare 941.*

```

# continua dall'esempio precedente

>>> print(response.headers.as_string()) ①
Date: Sun, 31 May 2009 19:23:06 GMT      ②
Server: Apache
Last-Modified: Sun, 31 May 2009 06:39:55 GMT ③
ETag: "bfe-93d9c4c0"                    ④
Accept-Ranges: bytes
Content-Length: 3070                     ⑤
Cache-Control: max-age=86400             ⑥
Expires: Mon, 01 Jun 2009 19:23:06 GMT
Vary: Accept-Encoding
Connection: close
Content-Type: application/xml

>>> data = response.read()              ⑦
>>> len(data)
3070

```

1. L'oggetto `response` restituito dalla funzione `urllib.request.urlopen()` contiene tutte le intestazioni HTTP che il server ha inviato nella risposta. Contiene anche i metodi per scaricare i dati effettivi, a cui arriveremo tra un minuto.
2. Il server vi dice quando si è occupato della vostra richiesta.
3. Questa risposta include un'intestazione Last-Modified.
4. Questa risposta include un'intestazione ETag.
5. La dimensione dei dati è 3070 byte. Notate quello che *manca* qui: un'intestazione `Content-encoding`. La vostra richiesta ha dichiarato di accettare solo dati non compressi (`Accept-encoding: identity`) e, come c'era da aspettarsi, questa risposta contiene dati non compressi.
6. Questa risposta include intestazioni di cache che affermano che questo feed può essere tenuto in memoria per 24 ore (86400 secondi).
7. E infine scarichiamo i dati effettivi invocando `response.read()`. Come potete vedere dal risultato della funzione `len()`, questa operazione scarica tutti i 3070 byte in una volta sola.

Come potete vedere, questo codice è già inefficiente: ha richiesto (e ricevuto) dati non compressi. So per certo che questo server supporta la compressione tramite gzip, ma la compressione HTTP è opzionale. Non

l'abbiamo richiesta e quindi non l'abbiamo ottenuta. Questo significa che stiamo scaricando 3070 byte mentre ne avremmo potuti scaricare solo 941. Cattivo cane, niente biscotto.

Ma aspettate, le cose peggiorano! Per vedere quanto questo codice sia davvero inefficiente, richiediamo lo stesso feed una seconda volta.

```
# continua dall'esempio precedente
>>> response2 = urlopen('http://diveintopython3.org/examples/feed.xml')
send: b'GET /examples/feed.xml HTTP/1.1
Host: diveintopython3.org
Accept-Encoding: identity
User-Agent: Python-urllib/3.1'
Connection: close
reply: 'HTTP/1.1 200 OK'
...ulteriori informazioni di debug omesse...
```

Notate qualcosa di peculiare in questa richiesta? Non è cambiata! È esattamente uguale alla prima richiesta. Nessun segno di intestazioni If-Modified-Since. Nessun segno di intestazioni If-None-Match. Nessun rispetto per le intestazioni di cache. Ancora nessuna compressione.

E cosa succede quando effettuate due volte la stessa richiesta? Ottenete due volte la stessa risposta.

```

# continua dall'esempio precedente

>>> print(response2.headers.as_string()) ①
Date: Mon, 01 Jun 2009 03:58:00 GMT
Server: Apache
Last-Modified: Sun, 31 May 2009 22:51:11 GMT
ETag: "bfe-255ef5c0"
Accept-Ranges: bytes
Content-Length: 3070
Cache-Control: max-age=86400
Expires: Tue, 02 Jun 2009 03:58:00 GMT
Vary: Accept-Encoding
Connection: close
Content-Type: application/xml

>>> data2 = response2.read()

>>> len(data2) ②
3070

>>> data2 == data ③
True

```

1. Il server sta ancora mandando la stessa schiera di intestazioni “intelligenti”: Cache-Control ed Expires per consentire l’uso della cache, Last-Modified ed ETag per abilitare il rilevamento della modifica più recente. Persino l’intestazione Vary: Accept-Encoding suggerisce che il server supporterebbe la compressione, se solo la chiedessimo. Ma non l’abbiamo fatto.
2. Ancora una volta, l’operazione di prelievo di questi dati scarica tutti i 3070 byte...
3. ...esattamente gli stessi 3070 byte che avete scaricato l’ultima volta.

HTTP è progettato per funzionare meglio di così. urllib parla HTTP come io parlo spagnolo — abbastanza bene da cavarmela improvvisando, ma non a sufficienza da tenere una conversazione. HTTP è una conversazione. È ora di sostituire urllib con una libreria in grado di parlare HTTP correntemente.



14.5. UNA INTRODUZIONE AD `httplib2`

Prima di usare `httplib2`, avrete bisogno di installarla. Visitate code.google.com/p/httplib2/ e scaricate l'ultima versione. `httplib2` è disponibile per Python 2.x e Python 3.x. Assicuratevi di prendere la versione per Python 3, che ha un nome simile a `httplib2-python3-0.5.0.zip`.

Estraete i contenuti dell'archivio, aprite una finestra di terminale e posizionatevi nella directory `httplib2` appena creata. Su Windows, aprite il menu Start, selezionate Esegui..., digitate `cmd.exe` e premete INVIO.

```
c:\Users\pilgrim\Downloads> dir
```

Il volume nell'unità C non ha etichetta.

Numero di serie del volume: DED5-B4F8

Directory di c:\Users\pilgrim\Downloads

```
28/07/2009  12:36    <DIR>          .
28/07/2009  12:36    <DIR>          ..
28/07/2009  12:36    <DIR>          httpplib2-python3-0.5.0
28/07/2009  12:33              18,997 httpplib2-python3-0.5.0.zip
                1 File              18.997 byte
                3 Directory        61.496.684.544 byte disponibili
```

```
c:\Users\pilgrim\Downloads> cd httpplib2-python3-0.5.0
```

```
c:\Users\pilgrim\Downloads\httpplib2-python3-0.5.0> c:\python31\python.exe setup.py install
```

eseguo install

eseguo build

eseguo build_py

eseguo install_lib

creo c:\python31\Lib\site-packages\httpplib2

copio build\lib\httpplib2\iri2uri.py -> c:\python31\Lib\site-packages\httpplib2

copio build\lib\httpplib2__init__.py -> c:\python31\Lib\site-packages\httpplib2

compilo c:\python31\Lib\site-packages\httpplib2\iri2uri.py in iri2uri.pyc

compilo c:\python31\Lib\site-packages\httpplib2__init__.py in __init__.pyc

eseguo install_egg_info

Scrivo c:\python31\Lib\site-packages\httpplib2-python3_0.5.0-py3.1.egg-info

Su Mac OS X, aprite l'applicazione Terminal.app che si trova nella vostra cartella /Applications/Utilities/.

Su Linux, lanciate l'applicazione Terminale, che di solito si trova nel vostro menu Applicazioni sotto la voce

Accessori o Strumenti di sistema.

```
you@localhost:~/Desktop$ unzip httpplib2-python3-0.5.0.zip
```

```
Archivio: httpplib2-python3-0.5.0.zip
```

```
decomprimo: httpplib2-python3-0.5.0/README
```

```
decomprimo: httpplib2-python3-0.5.0/setup.py
```

```
decomprimo: httpplib2-python3-0.5.0/PKG-INFO
```

```
decomprimo: httpplib2-python3-0.5.0/httpplib2/__init__.py
```

```
decomprimo: httpplib2-python3-0.5.0/httpplib2/iri2uri.py
```

```
you@localhost:~/Desktop$ cd httpplib2-python3-0.5.0/
```

```
you@localhost:~/Desktop/httpplib2-python3-0.5.0$ sudo python3 setup.py install
```

```
eseguo install
```

```
eseguo build
```

```
eseguo build_py
```

```
creo build
```

```
creo build/lib.linux-x86_64-3.1
```

```
creo build/lib.linux-x86_64-3.1/httpplib2
```

```
copio httpplib2/iri2uri.py -> build/lib.linux-x86_64-3.1/httpplib2
```

```
copio httpplib2/__init__.py -> build/lib.linux-x86_64-3.1/httpplib2
```

```
eseguo install_lib
```

```
creo /usr/local/lib/python3.1/dist-packages/httpplib2
```

```
copio build/lib.linux-x86_64-3.1/httpplib2/iri2uri.py -> /usr/local/lib/python3.1/dist-packages/httpplib2
```

```
copio build/lib.linux-x86_64-3.1/httpplib2/__init__.py -> /usr/local/lib/python3.1/dist-packages/httpplib2
```

```
compilo /usr/local/lib/python3.1/dist-packages/httpplib2/iri2uri.py in iri2uri.pyc
```

```
compilo /usr/local/lib/python3.1/dist-packages/httpplib2/__init__.py in __init__.pyc
```

```
eseguo install_egg_info
```

```
Scrivo /usr/local/lib/python3.1/dist-packages/httpplib2-python3_0.5.0.egg-info
```

Per utilizzare httpplib2, create un'istanza della classe httpplib2.Http.


```

>>> import httpplib2

>>> h = httpplib2.Http('.cache') ①

>>> response, content = h.request('http://diveintopython3.org/examples/feed.xml') ②

>>> response.status ③

200

>>> content[:52] ④


b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="

>>> len(content)

3070

```

1. L'interfaccia principale al modulo `httpplib2` è l'oggetto `Http`. Per ragioni che vedrete nella prossima sezione, dovreste sempre passare un nome di directory quando create un oggetto `Http`. La directory non deve per forza esistere, `httpplib2` la creerà se è necessario.
2. Una volta che avete un oggetto `Http`, recuperare dati è tanto semplice quanto invocare il metodo `request()` con l'indirizzo dei dati che volete. Questo metodo emetterà una richiesta HTTP GET per quell'URL. (Più avanti in questo capitolo vedrete come emettere altre richieste HTTP, come POST.)
3. Il metodo `request()` restituisce due valori. Il primo è un oggetto `httpplib2.Response`, che contiene tutte le intestazioni HTTP restituite dal server. Per esempio, il codice di stato 200 memorizzato nell'attributo `status` indica che la richiesta ha avuto successo.
4. La variabile `content` contiene i dati effettivi che sono stati restituiti dal server HTTP. I dati vengono restituiti come un oggetto bytes, non una stringa. Se li volete sotto forma di stringa dovete determinarne la codifica di carattere ed effettuare da voi la conversione.

 Avrete probabilmente bisogno di un solo oggetto `httpplib2.Http`. Esistono valide ragioni per crearne più di uno, ma dovreste farlo solo se sapete perché ne avete bisogno. “Devo richiedere dati da due URL differenti” non è una ragione valida. Riutilizzate l'oggetto `Http` e invocate semplicemente il metodo `request()` due volte.

14.5.1. UNA BREVE DIGRESSIONE PER SPIEGARE PERCHÉ `httplib2` RESTITUISCE `BYTE` INVECE DI `STRINGHE`

Byte. Stringhe. Che sofferenza. Perché `httplib2` non può “semplicemente” fare la conversione per voi? Be’, è complicato, perché le regole per determinare la codifica di carattere sono specifiche per il tipo di risorsa che state richiedendo. Come potrebbe fare `httplib2` a sapere che tipo di risorsa state richiedendo? Di solito, il tipo è elencato nell’intestazione `HTTP Content-Type`, ma questa è una caratteristica opzionale di `HTTP` e non tutti i server `HTTP` la supportano. Se quell’intestazione non è presente nella risposta `HTTP`, tocca al client indovinare. Questa operazione viene comunemente chiamata “content sniffing” (letteralmente, annusare il contenuto) e non è mai perfetta.

Se sapete che tipo di risorsa vi aspettate (in questo caso, un documento XML), forse potreste “semplicemente” passare l’oggetto bytes restituito alla funzione `xml.etree.ElementTree.parse()`. Questo funzionerà purché il documento XML includa informazioni sulla propria codifica di carattere (come accade in questo caso), ma questa è una caratteristica opzionale e non tutti i documenti XML lo fanno. Se un documento XML non include informazioni di codifica, il client è tenuto a controllare il protocollo di trasporto a cui sono allegati i dati — cioè l’intestazione `HTTP Content-Type`, che può includere un parametro `charset`.

Ma le cose vanno anche peggio di così. Ora le informazioni sulla codifica di carattere possono trovarsi in due posti: all’interno del documento XML e nell’intestazione `HTTP Content-Type`. Se l’informazione è in *entrambi* i posti, quale deve essere ritenuta valida? Secondo la [RFC 3023](#) (vi giuro che non me lo sto inventando), se il tipo di contenuto fornito nell’intestazione `HTTP Content-Type` è `application/xml`, `application/xml-dtd`, `application/xml-external-parsed-entity`, o un qualsiasi sottotipo di `application/xml` come per esempio `application/atom+xml` o `application/rss+xml` o persino `application/rdf+xml`, allora la codifica è

[Maglietta “lo
supporto la RFC
3023”]

1. la codifica data nel parametro `charset` dell’intestazione `HTTP Content-Type`, oppure
2. la codifica data nell’attributo `encoding` della dichiarazione XML all’interno del documento, oppure
3. `UTF-8`.

D'altra parte, se il tipo di contenuto dato nell'intestazione HTTP Content-Type è text/xml, text/xml-external-parsed-entity, o un sottotipo come per esempio text/QualsiasiCosa+xml, allora l'attributo di codifica della dichiarazione XML nel documento viene completamente ignorato e la codifica è

1. la codifica data nel parametro charset dell'intestazione HTTP Content-Type, oppure
2. US-ASCII.

E questo è solo per i documenti XML. Per i documenti HTML, i browser web hanno costruito regole per il content sniffing [PDF] talmente bizantine che stiamo ancora cercando di capire come funzionano.

“Le patch sono benvenute.”

14.5.2. COME http lib2 GESTISCE LA CACHE

Ricordate quando, nella sezione precedente, ho detto che dovrete sempre creare un oggetto http lib2.Http con un nome di directory? Il motivo è la cache.

```
# continua dall'esempio precedente

>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed.xml') ①
>>> response2.status ②
200
>>> content2[:52] ③
b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="
>>> len(content2)
3070
```

1. Questo non dovrebbe essere terribilmente sorprendente. È la stessa cosa che avete fatto l'ultima volta, senonché state mettendo il risultato in due nuove variabili.
2. Il valore del codice di stato HTTP memorizzato in status è ancora 200, proprio come l'ultima volta.
3. Anche il contenuto scaricato è lo stesso dell'ultima volta.

E quindi... chi se ne importa? Uscite dalla shell interattiva di Python e rilanciatela con una nuova sessione, e vi farò vedere.

```

# NON continua dall'esempio precedente!
# Per favore uscite dalla shell interattiva
# e lanciatene una nuova.
>>> import httpplib2
>>> httpplib2.debuglevel = 1 ①
>>> h = httpplib2.Http('.cache') ②
>>> response, content = h.request('http://diveintopython3.org/examples/feed.xml') ③
>>> len(content) ④
3070
>>> response.status ⑤
200
>>> response.fromcache ⑥
True

```

1. Attiviamo il debug e vediamo cosa viene trasmesso attraverso la rete. Questo è l'equivalente per httpplib2 dell'attivazione del debug in `http.client`. httpplib2 stamperà tutti i dati inviati al server e alcune informazioni chiave restituite dal server.
2. Create un oggetto `httpplib2.Http` con lo stesso nome di directory precedente.
3. Richiedete lo stesso URL di prima. *Sembra che non accada nulla*. Più precisamente, nulla viene inviato al server e nulla viene restituito dal server. Non c'è assolutamente alcuna attività di rete.
4. Tuttavia abbiamo “ricevuto” alcuni dati — in effetti, li abbiamo ricevuti tutti.
5. Abbiamo anche “ricevuto” un codice di stato HTTP che indica che la “richiesta” ha avuto successo.
6. Qui sta il punto: questa “risposta” è stata generata dalla cache locale di httpplib2. Quel nome di directory che avete passato nel creare l'oggetto `httpplib2.Http` — quella directory conserva la cache di httpplib2 per tutte le operazioni che ha mai effettuato.

☞ Se volete attivare le informazioni di debug di `httplib2`, dovete impostare una costante a livello di modulo (`httplib2.debuglevel`) e poi creare un nuovo oggetto `httplib2.Http`. Se volete disattivare le informazioni di debug, dovete modificare la stessa costante a livello di modulo e poi creare un nuovo oggetto `httplib2.Http`.

Avete già richiesto in precedenza i dati a questo URL. Quella richiesta ha avuto successo (status: 200). Quella risposta includeva non solo i dati del feed, ma anche un insieme di intestazioni di cache che dicevano a chiunque fosse in ascolto che poteva tenere in cache questa risorsa per 24 ore. (Cache-Control: max-age=86400, che sono 24 ore misurate in secondi). `httplib2` comprende e rispetta queste intestazioni di cache, e ha memorizzato la risposta precedente nella directory `.cache` (il cui nome avete passato quando avete creato l'oggetto `Http`). Quella informazione in cache non è ancora scaduta, quindi la seconda volta che richiedete i dati a questo URL `httplib2` semplicemente restituisce il risultato in cache senza nemmeno utilizzare la rete.

Ho detto “semplicemente”, ma ovviamente c'è molta complessità nascosta dietro questa semplicità. `httplib2` gestisce l'uso della cache da parte di HTTP in maniera *automatica* e *predefinita*. Casomai per qualche ragione aveste bisogno di sapere se una risposta proveniva dalla cache, potete controllare l'attributo `response.fromcache`. Altrimenti, tutto funziona e basta.

Ora, supponete di avere i dati nella cache, ma di voler aggirare la cache e riottenerli dal server remoto. I browser talvolta lo fanno se l'utente lo richiede esplicitamente. Per esempio, premere F5 aggiorna la pagina corrente, ma premere Ctrl+F5 aggira la cache e riottiene la pagina corrente dal server remoto. Potreste pensare: “Oh, cancellerò semplicemente i dati dalla mia cache locale, poi li richiederò di nuovo.” Potreste farlo, ma ricordate che potrebbero esserci più parti coinvolte anziché solo voi e il server remoto. Cosa mi dite di quei server proxy intermedi? Quelli sono completamente al di là del vostro controllo, potrebbero

*Cosa viene
trasMESSO
attraverso la
rete? Proprio
nulla.*

ancora avere quei dati nella propria cache e ve li restituiranno tranquillamente perché (per quanto li riguarda) la loro cache è ancora valida.

Invece di manipolare la vostra cache locale e sperare per il meglio, dovrete usare le caratteristiche di HTTP per assicurarvi che la vostra richiesta raggiunga effettivamente il server remoto.

```

# continua dall'esempio precedente
>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed.xml',
...     headers={'cache-control': 'no-cache'}) ①
connect: (diveintopython3.org, 80) ②
send: b'GET /examples/feed.xml HTTP/1.1
Host: diveintopython3.org
user-agent: Python-httpplib2/$Rev: 259 $
accept-encoding: deflate, gzip
cache-control: no-cache'
reply: 'HTTP/1.1 200 OK'
...ulteriori informazioni di debug omesse...
>>> response2.status
200
>>> response2.fromcache ③
False
>>> print(dict(response2.items())) ④
{'status': '200',
 'content-length': '3070',
 'content-location': 'http://diveintopython3.org/examples/feed.xml',
 'accept-ranges': 'bytes',
 'expires': 'Wed, 03 Jun 2009 00:40:26 GMT',
 'vary': 'Accept-Encoding',
 'server': 'Apache',
 'last-modified': 'Sun, 31 May 2009 22:51:11 GMT',
 'connection': 'close',
 '-content-encoding': 'gzip',
 'etag': '"bfe-255ef5c0"',
 'cache-control': 'max-age=86400',
 'date': 'Tue, 02 Jun 2009 00:40:26 GMT',
 'content-type': 'application/xml'}
```

- I. `httplib2` vi consente di aggiungere intestazioni HTTP arbitrarie a qualsiasi richiesta in uscita. Per aggirare tutte le cache (non solo quella sul vostro disco locale, ma anche tutti i proxy di cache tra voi e il server remoto), aggiungete un'intestazione `no-cache` nel dizionario `headers`.

2. Ora vedete che `httplib2` inizia una richiesta di rete. `httplib2` comprende e rispetta le intestazioni di cache *in entrambe le direzioni* — come parte della risposta in arrivo e *come parte della richiesta in partenza*. Il modulo ha notato che avete aggiunto l'intestazione `no-cache`, quindi ha aggirato la propria cache locale e non ha avuto altra scelta se non quella di utilizzare la rete per richiedere i dati.
3. Questa risposta *non* è stata generata dalla vostra cache locale. Lo sapevate, naturalmente, perché avete visto le informazioni di debug sulla richiesta in partenza. Ma è piacevole averlo programmaticamente verificato.
4. La richiesta ha avuto successo, avete nuovamente scaricato l'intero feed dal server remoto. Naturalmente, il server ha anche spedito una serie completa di intestazioni HTTP insieme ai dati del feed, comprese le intestazioni di cache che `httplib2` utilizza per aggiornare la propria cache locale nella speranza di evitare l'accesso alla rete la *prossima* volta che richiedete questo feed. Ogni caratteristica della gestione della cache in HTTP è progettata per massimizzare l'uso della cache e minimizzare l'accesso alla rete. Anche se avete aggirato la cache questa volta, il server remoto apprezzerrebbe davvero che voi manteneste in cache il risultato per la prossima volta.

14.5.3. COME `httplib2` GESTISCE LE INTESTAZIONI `Last-Modified` ED `ETag`

Le intestazioni di cache `Cache-Control` ed `Expires` sono chiamate *indicatori di freschezza* e dicono alle cache in termini certi che potete evitare completamente ogni accesso alla rete fino a quando la cache scade. Questo è esattamente il comportamento che avete visto nella sezione precedente: dato un indicatore di freschezza, `httplib2` *non genera un singolo byte di attività di rete* per prelevare i dati mantenuti in cache (a meno che voi non aggiariate la cache in maniera esplicita, naturalmente).

Ma cosa succede nel caso in cui i dati *potrebbero* essere cambiati, ma non lo sono? HTTP definisce le intestazioni `Last-Modified` ed `ETag` proprio a questo scopo. Queste intestazioni sono chiamate *validatori*. Se la cache locale non è più fresca, un client può spedire i validatori insieme alla prossima richiesta per vedere se i dati sono effettivamente cambiati. Se i dati non sono cambiati, il server risponde con un codice di stato 304 e *nessun dato*. Quindi c'è ancora un viaggio attraverso la rete, ma finite per scaricare meno byte.


```

>>> import httpplib2
>>> httpplib2.debuglevel = 1
>>> h = httpplib2.Http('.cache')
>>> response, content = h.request('http://diveintopython3.org/') ①
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httpplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'

>>> print(dict(response.items())) ②
{'-content-encoding': 'gzip',
 'accept-ranges': 'bytes',
 'connection': 'close',
 'content-length': '6657',
 'content-location': 'http://diveintopython3.org/',
 'content-type': 'text/html',
 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
 'etag': '"7f806d-1a01-9fb97900"', 'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
 'server': 'Apache',
 'status': '200',
 'vary': 'Accept-Encoding,User-Agent'}

>>> len(content) ③
6657

```

1. Invece di un feed, questa volta scaricheremo la pagina iniziale del sito, che è in HTML. Dato che questa è la prima volta che richiedete questa pagina, httpplib2 ha poco con cui lavorare e invia un insieme minimo di intestazioni con la richiesta.
2. La risposta contiene una moltitudine di intestazioni HTTP... ma nessuna informazione di cache. Tuttavia, include entrambe le intestazioni ETag e Last-Modified.
3. Al momento in cui ho realizzato questo esempio, la pagina era di 6657 byte. Probabilmente ora è cambiata, ma non dovete preoccuparvi di questo.

```

# continua dall'esempio precedente

>>> response, content = h.request('http://diveintopython3.org/') ①
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org

if-none-match: "7f806d-1a01-9fb97900" ②
if-modified-since: Tue, 02 Jun 2009 02:51:48 GMT ③
accept-encoding: deflate, gzip
user-agent: Python-httpplib2/$Rev: 259 $'
reply: 'HTTP/1.1 304 Not Modified' ④

>>> response.fromcache ⑤
True

>>> response.status ⑥
200

>>> response.dict['status'] ⑦
'304'

>>> len(content) ⑧
6657

```

1. Avete richiesto ancora la stessa pagina con lo stesso oggetto `Http` (e la stessa cache locale).
2. `httplib2` spedisce il validatore `ETag` al server nell'intestazione `If-None-Match`.
3. `httplib2` spedisce al server anche il validatore `Last-Modified` nell'intestazione `If-Modified-Since`.
4. Il server ha esaminato questi validatori, ha esaminato la pagina che avete richiesto e ha determinato che la pagina non è cambiata dall'ultima volta che l'avete richiesta, quindi risponde con un codice di stato `304` e *nessun dato*.
5. Tornando al client, `httplib2` nota il codice di stato `304` e carica il contenuto della pagina dalla propria cache.
6. Questo potrebbe confondervi un poco. In realtà ci sono *due* codici di stato — `304` (restituito dal server questa volta, che induce `httplib2` a guardare nella propria cache) e `200` (restituito dal server *l'ultima volta* e memorizzato nella cache di `httplib2` insieme ai dati della pagina). `response.status` contiene lo stato proveniente dalla cache.
7. Se volete lo stato effettivo restituito dal server, potete ottenerlo guardando in `response.dict`, che è un dizionario delle reali intestazioni restituite dal server.

8. Tuttavia, i dati vi vengono resi disponibili ancora nella variabile `content`. Generalmente, non avete bisogno di sapere perché una risposta è stata servita dalla cache. (Potete persino ignorare il fatto che sia stata servita dalla cache, e anche questo va bene. `http1b2` è abbastanza intelligente da lasciarvi agire in modo stupido.) Nel momento in cui il metodo `request()` restituisce il controllo al chiamante, `http1b2` ha già aggiornato la propria cache e vi ha restituito i dati.

14.5.4. COME `http2lib` GESTISCE LA COMPRESSIONE

HTTP supporta diversi tipi di compressione; i due tipi più comuni sono `gzip` e `deflate`. `http1b2` li supporta entrambi.

*“Facciamo
musica di
entrambi i
generi,
country E
western.”*

```

>>> response, content = h.request('http://diveintopython3.org/')
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httpplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'
>>> print(dict(response.items()))
{'-content-encoding': 'gzip',
 'accept-ranges': 'bytes',
 'connection': 'close',
 'content-length': '6657',
 'content-location': 'http://diveintopython3.org/',
 'content-type': 'text/html',
 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
 'etag': '"7f806d-1a01-9fb97900"',
 'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
 'server': 'Apache',
 'status': '304',
 'vary': 'Accept-Encoding,User-Agent'}

```

①

②

1. Ogni volta che `httplib2` invia una richiesta, include un'intestazione `Accept-Encoding` per dire al server che è in grado di gestire la compressione sia di tipo `deflate` che di tipo `gzip`.
2. In questo caso, il server ha risposto con un carico utile compresso tramite `gzip`. Nel momento in cui il metodo `request()` termina, `httplib2` ha già decompresso il corpo della risposta e lo ha piazzato nella variabile `content`. Casomai foste curiosi di sapere se la risposta era compressa oppure no, potete controllare `response['-content-encoding']`; altrimenti, non preoccupatevi di questo.

14.5.5. COME `httplib2` GESTISCE LE REDIREZIONI

HTTP definisce due tipi di redirezioni: temporanee e permanenti. Non c'è niente di speciale da fare con le redirezioni temporanee, tranne seguirle, cosa che `httplib2` esegue automaticamente.

```

>>> import httpplib2
>>> httpplib2.debuglevel = 1
>>> h = httpplib2.Http('.cache')
>>> response, content = h.request('http://diveintopython3.org/examples/feed-302.xml') ①
connect: (diveintopython3.org, 80)
send: b'GET /examples/feed-302.xml HTTP/1.1' ②
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httpplib2/$Rev: 259 $'
reply: 'HTTP/1.1 302 Found' ③
send: b'GET /examples/feed.xml HTTP/1.1' ④
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httpplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'

```

1. Non c'è alcun feed a questo URL. Ho impostato il mio server in modo da emettere una redirezione temporanea verso l'indirizzo corretto.
2. Questa è la richiesta.
3. E questa è la risposta: 302 Found. Qui non viene mostrata, ma questa risposta include anche un'intestazione Location che punta al vero URL.
4. httpplib2 torna immediatamente sui suoi passi e “segue” la redirezione emettendo un'altra richiesta per l'URL contenuto nell'intestazione Location: http://diveintopython3.org/examples/feed.xml

“Seguire” una redirezione non è niente di più di ciò che viene mostrato in questo esempio. httpplib2 invia una richiesta per l'URL che avete chiesto. Il server ribatte con una risposta che dice: “No, no, guarda qui invece.” httpplib2 invia un'altra richiesta per il nuovo URL.

```
# continua dall'esempio precedente

>>> response ①
{'status': '200',
 'content-length': '3070',
 'content-location': 'http://diveintopython3.org/examples/feed.xml', ②
 'accept-ranges': 'bytes',
 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
 'vary': 'Accept-Encoding',
 'server': 'Apache',
 'last-modified': 'Wed, 03 Jun 2009 02:20:15 GMT',
 'connection': 'close',
 '-content-encoding': 'gzip', ③
 'etag': '"bfe-4cbbf5c0"',
 'cache-control': 'max-age=86400', ④
 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
 'content-type': 'application/xml'}
```

1. I dati memorizzati nella variabile `response` che ottenete da questa singola chiamata al metodo `request()` sono la risposta dall'URL finale.
2. `httplib2` aggiunge l'URL finale al dizionario `response`, come valore per la chiave `content-location`. Questa non è un'intestazione proveniente dal server, ma è specifica per `httplib2`.
3. Incidentalmente, questo feed è compresso.
4. E memorizzabile in cache. (Questo è importante, come vedrete fra un minuto.)

La risposta contenuta nella variabile `response` che avete ottenuto vi dà informazioni sull'URL *finale*. E se voleste informazioni sugli URL intermedi, quelli che alla fine vi hanno rediretto all'URL finale? `httplib2` vi permette di ottenere queste informazioni.

```

# continua dall'esempio precedente

>>> response.previous ①
{'status': '302',
 'content-length': '228',
 'content-location': 'http://diveintopython3.org/examples/feed-302.xml',
 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
 'server': 'Apache',
 'connection': 'close',
 'location': 'http://diveintopython3.org/examples/feed.xml',
 'cache-control': 'max-age=86400',
 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
 'content-type': 'text/html; charset=iso-8859-1'}

>>> type(response) ②
<class 'httplib2.Response'>

>>> type(response.previous)
<class 'httplib2.Response'>

>>> response.previous.previous ③
>>>

```

1. L'attributo `response.previous` mantiene un riferimento al precedente oggetto risposta che `httplib2` ha seguito per ottenere l'oggetto risposta attuale.
2. Sia `response` che `response.previous` sono oggetti `httplib2.Response`.
3. Questo significa che potete controllare `response.previous.previous` per seguire la catena di redirezioni facendo ulteriori passi indietro. (Scenario: un URL dirige verso un secondo URL che dirige verso un terzo URL. Potrebbe accadere!) In questo caso, abbiamo già raggiunto l'inizio della catena di redirezioni, quindi l'attributo vale `None`.

Cosa succede se effettuate ancora una richiesta allo stesso URL?

```

# continua dall'esempio precedente

>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed-302.xml') ①
connect: (diveintopython3.org, 80)

send: b'GET /examples/feed-302.xml HTTP/1.1 ②
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httpplib2/$Rev: 259 $'

reply: 'HTTP/1.1 302 Found' ③

>>> content2 == content ④
True

```

1. Stesso URL, stesso oggetto `httplib2.Http` (e quindi stessa cache).
2. La risposta 302 non è stata memorizzata in cache, quindi `httplib2` invia un'altra richiesta per lo stesso URL.
3. Ancora una volta, il server risponde con un 302. Ma notate che *non* è successo: non c'è mai stata una seconda richiesta per l'URL finale, `http://diveintopython3.org/examples/feed.xml`. Quella risposta è stata memorizzata in cache (ricordatevi l'intestazione `Cache-Control` che avevate visto nell'esempio precedente). Una volta che `httplib2` ha ricevuto il codice 302 Found, *ha controllato la propria cache prima di emettere un'altra richiesta*. La cache conteneva una copia fresca di `http://diveintopython3.org/examples/feed.xml`, quindi non c'era nessun bisogno di riottenerla.
4. Nel momento in cui la sua esecuzione termina, il metodo `request()` ha già letto i dati del feed dalla cache e li ha restituiti. Naturalmente, sono gli stessi dati che avevate ricevuto l'ultima volta.

In altre parole, non dovete fare niente di speciale per le redirezioni temporanee. `httplib2` le seguirà automaticamente, e il fatto che un URL reindirizzi a un altro non ha alcun rapporto con il supporto da parte di `httplib2` per la compressione, la cache, gli ETag, o qualsiasi altra caratteristica di HTTP.

Le redirezioni permanenti sono altrettanto semplici.


```
# continua dall'esempio precedente

>>> response, content = h.request('http://diveintopython3.org/examples/feed-301.xml') ①
connect: (diveintopython3.org, 80)
send: b'GET /examples/feed-301.xml HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httpplib2/$Rev: 259 $'
reply: 'HTTP/1.1 301 Moved Permanently' ②

>>> response.fromcache ③
True
```

1. Ancora una volta, in realtà questo URL non esiste. Ho impostato il mio server in modo da emettere una redirezione permanente verso `http://diveintopython3.org/examples/feed.xml`.
2. Ed eccola qui: codice di stato 301. Ma ancora, notate cosa *non* è successo: non c'è stata alcuna richiesta verso l'URL rediretto. Perché no? Perché è già memorizzato in cache localmente.
3. `httplib2` ha “seguito” la redirezione dritto nella propria cache.

Ma aspettate! C'è di più!

```
# continua dall'esempio precedente

>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed-301.xml') ①
>>> response2.fromcache ②
True
>>> content2 == content ③
True
```

1. Ecco una differenza tra le redirezioni temporanee e permanenti: una volta che `httplib2` segue una redirezione permanente, tutte le ulteriori richieste per quell'URL verranno trasparentemente riscritte per dirigersi verso l'URL obiettivo *senza utilizzare la rete per l'URL originale*. Ricordate, il debug è ancora attivo, tuttavia non risulta alcuna attività di rete.
2. Sì, questa risposta è stata recuperata dalla cache locale.
3. Sì, avete ottenuto l'intero feed (dalla cache).

HTTP. Funziona.



14.6. OLTRE HTTP GET

I servizi web HTTP non sono limitati alle richieste GET. E se voleste creare qualcosa di nuovo? Ogni volta che inviate un commento a una discussione su un forum, aggiornate il vostro weblog, pubblicate il vostro stato su un servizio di microblogging come [Twitter](#) o [Identi.ca](#), state probabilmente già usando HTTP POST.

Sia Twitter che Identi.ca offrono una semplice API basata su HTTP per pubblicare e aggiornare il vostro stato tramite messaggi non più lunghi di 140 caratteri. Diamo un'occhiata alla [documentazione della API di Identi.ca](#) per l'aggiornamento del vostro stato.

Metodo della API REST di Identi.ca: statuses/update

Aggiorna lo stato dell'utente autenticato. Richiede il parametro `status` specificato sotto. La richiesta deve essere di tipo POST.

URL

`https://identi.ca/api/statuses/update.formato`

Formati

`xml, json, rss, atom`

Metodi HTTP

`POST`

Richiede autenticazione

`vero`

Parametri

`status`. Obbligatorio. Il testo del vostro aggiornamento di stato. Codificatelo come URL se necessario.

Come funziona questo metodo? Per pubblicare un nuovo messaggio su Identi.ca avete bisogno di emettere una richiesta HTTP POST verso `http://identi.ca/api/statuses/update.formato`. (Il formato non è parte dell'URL, ma dovete sostituirlo con il formato dei dati che volete farvi restituire dal server in risposta alla vostra richiesta. Quindi, se volete una risposta in XML dovreste spedire la richiesta a `https://identi.ca/api/statuses/update.xml`.) La richiesta deve includere un parametro chiamato `status` che contiene il testo del vostro aggiornamento di stato. E la richiesta deve essere autenticata.

Autenticata? Certamente. Per aggiornare il vostro stato su Identi.ca dovete provare chi siete. Identi.ca non è un wiki, solo voi potete aggiornare il vostro stato. Identi.ca utilizza la Basic Authentication di HTTP (alias RFC 2617) via SSL per fornire un'autenticazione sicura ma facile da usare. `httplib2` supporta la Basic Authentication sia via SSL che via HTTP, quindi questa parte è facile.

Una richiesta POST è diversa da una richiesta GET perché include un *carico utile*. Il carico utile sono i dati che volete inviare al server. L'unica informazione *obbligatoria* che questo metodo della API richiede è `status`, e dovrebbe essere *codificata come URL*. Questa codifica è un formato di serializzazione molto semplice che prende un insieme di coppie chiave-valore (cioè un dizionario) e lo trasforma in una stringa.

```
>>> from urllib.parse import urlencode ①
>>> data = {'status': 'Aggiornamento di prova da Python 3'} ②
>>> urlencode(data) ③
'status=Aggiornamento+di+prova+da+Python+3'
```


1. Python include una funzione di utilità per codificare come URL un dizionario: `urllib.parse.urlencode()`.
2. Questo è il tipo di dizionario di cui la API di Identi.ca ha bisogno. Contiene una chiave `status` il cui valore è il testo di un singolo aggiornamento di stato.
3. Questa è la stringa codificata come URL. Questo è il *carico utile* che verrà inviato attraverso la rete al server della API di Identi.ca nella vostra richiesta HTTP POST.

```

>>> from urllib.parse import urlencode
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')
>>> data = {'status': 'Aggiornamento di prova da Python 3'}
>>> h.add_credentials('diveintomark', 'MIA_PASSWORD_SEGRETA', 'identi.ca') ①
>>> resp, content = h.request('https://identi.ca/api/statuses/update.xml',
...     'POST', ②
...     urlencode(data), ③
...     headers={'Content-Type': 'application/x-www-form-urlencoded'}) ④

```

1. Questo è il modo in cui `httplib2` gestisce l'autenticazione. Memorizzate il vostro nome utente e la vostra password con il metodo `add_credentials()`. Quando `httplib2` prova a emettere la richiesta, il server risponderà con un codice di stato 401 Unauthorized ed elencherà quali metodi di autenticazione supporta (nell'intestazione `WWW-Authenticate`). `httplib2` costruirà automaticamente un'intestazione `Authorization` e richiederà nuovamente l'URL.
2. Il secondo parametro è il tipo di richiesta HTTP, POST in questo caso.
3. Il terzo parametro è il *carico utile* da spedire al server. Stiamo inviando il dizionario codificato come URL con il messaggio di stato.
4. Infine, abbiamo bisogno di dire al server che il carico utile contiene dati codificati come URL.

 Il terzo parametro del metodo `add_credentials()` è il dominio in cui le credenziali sono valide. Dovreste sempre specificarlo! Se lasciate fuori il dominio e più tardi riutilizzate l'oggetto `httplib2.Http` su un sito autenticato differente, `httplib2` potrebbe far trapelare il nome utente e la password di un sito all'altro sito.

Questo è quello che viene trasmesso attraverso la rete:

```

# continua dall'esempio precedente
send: b'POST /api/statuses/update.xml HTTP/1.1
Host: identi.ca
Accept-Encoding: identity
Content-Length: 41
content-type: application/x-www-form-urlencoded
user-agent: Python-httpplib2/$Rev: 259 $

status=Aggiornamento+di+prova+da+Python+3'

reply: 'HTTP/1.1 401 Unauthorized' ①

send: b'POST /api/statuses/update.xml HTTP/1.1 ②
Host: identi.ca
Accept-Encoding: identity
Content-Length: 41
content-type: application/x-www-form-urlencoded
authorization: Basic HASH_SEGRETO_COSTRUITO_DA_HTTPLIB2 ③
user-agent: Python-httpplib2/$Rev: 259 $

status=Aggiornamento+di+prova+da+Python+3'

reply: 'HTTP/1.1 200 OK' ④

```

1. Dopo la prima richiesta, il server risponde con un codice di stato 401 Unauthorized. httpplib2 non invierà le intestazioni di autenticazione a meno che il server non le chieda esplicitamente. Questo è il modo in cui il server le chiede.
2. httpplib2 torna immediatamente sui suoi passi e richiede lo stesso URL una seconda volta.
3. Questa volta include il nome utente e la password che avete aggiunto tramite il metodo `add_credentials()`.
4. Ha funzionato!

Cos'è che il server spedisce indietro dopo una richiesta che ha successo? Questo dipende interamente dalla API del servizio web. In alcuni protocolli (come il Protocollo di Pubblicazione Atom) il server risponde con un codice di stato 201 Created e l'ubicazione della risorsa appena creata nell'intestazione Location. Identi.ca restituisce un codice di stato 200 OK e un documento XML contenente informazioni sulla risorsa appena creata.

```

# continua dall'esempio precedente

>>> print(content.decode('utf-8')) ①
<?xml version="1.0" encoding="UTF-8"?>
<status>

<text>Aggiornamento di prova da Python 3</text> ②
<truncated>>false</truncated>

<created_at>Wed Jun 10 03:53:46 +0000 2009</created_at>
<in_reply_to_status_id></in_reply_to_status_id>
<source>api</source>

<id>5131472</id> ③
<in_reply_to_user_id></in_reply_to_user_id>
<in_reply_to_screen_name></in_reply_to_screen_name>
<favorited>>false</favorited>

<user>

  <id>3212</id>
  <name>Mark Pilgrim</name>
  <screen_name>diveintomark</screen_name>
  <location>27502, US</location>
  <description>autore di libri tecnici, marito, padre</description>
  <profile_image_url>http://avatar.identi.ca/3212-48-20081216000626.png</profile_image_url>
  <url>http://diveintomark.org/</url>
  <protected>>false</protected>
  <followers_count>329</followers_count>
  <profile_background_color></profile_background_color>
  <profile_text_color></profile_text_color>
  <profile_link_color></profile_link_color>
  <profile_sidebar_fill_color></profile_sidebar_fill_color>
  <profile_sidebar_border_color></profile_sidebar_border_color>
  <friends_count>2</friends_count>
  <created_at>Wed Jul 02 22:03:58 +0000 2008</created_at>
  <favourites_count>30768</favourites_count>
  <utc_offset>0</utc_offset>
  <time_zone>UTC</time_zone>
  <profile_background_image_url></profile_background_image_url>

```

```
<profile_background_tile>false</profile_background_tile>
<statuses_count>122</statuses_count>
<following>false</following>
<notifications>false</notifications>
</user>
</status>
```

1. Ricordatevi che i dati restituiti da `httplib2` sono sempre `byte`, non stringhe. Per convertirli in una stringa dovete decodificarli utilizzando la codifica di carattere corretta. I metodi della API di `Identi.ca` restituiscono sempre i risultati in UTF-8, quindi questa parte è facile.
2. Ecco il testo del messaggio di stato che abbiamo appena pubblicato.
3. Ecco l'identificatore unico per il nuovo messaggio di stato. `Identi.ca` lo usa per costruire un URL in modo da visualizzare il messaggio sul web.

Ed eccolo qui:





14.7. OLTRE HTTP POST

HTTP non è limitato a GET e POST. Questi sono certamente i tipi più comuni di richieste, specialmente nei browser web. Ma le API di un servizio web possono andare oltre GET e POST, e `httplib2` è pronta.

```
# continua dall'esempio precedente
>>> from xml.etree import ElementTree as etree
>>> tree = etree.fromstring(content) ①
>>> status_id = tree.findtext('id') ②
>>> status_id
'5131472'
>>> url = 'https://identi.ca/api/statuses/destroy/{0}.xml'.format(status_id) ③
>>> resp, deleted_content = h.request(url, 'DELETE') ④
```

1. Il server ha restituito un documento XML, giusto? Voi sapete come riconoscere un documento XML.
2. Il metodo `findtext()` trova la prima istanza dell'espressione data e ne estrae il contenuto testuale. In questo caso, stavamo giusto cercando un elemento `<id>`.
3. Sulla base del contenuto testuale dell'elemento `<id>` possiamo costruire un URL per cancellare il messaggio di stato che abbiamo appena pubblicato.
4. Per cancellare un messaggio vi basta emettere una richiesta HTTP DELETE verso quell'URL.

Questo è quello che viene trasmesso attraverso la rete:


```

send: b'DELETE /api/statuses/destroy/5131472.xml HTTP/1.1 ①
Host: identi.ca
Accept-Encoding: identity
user-agent: Python-httpplib2/$Rev: 259 $

,

reply: 'HTTP/1.1 401 Unauthorized' ②
send: b'DELETE /api/statuses/destroy/5131472.xml HTTP/1.1 ③
Host: identi.ca
Accept-Encoding: identity
authorization: Basic HASH_SEGRETO_COSTRUITO_DA_HTTPLIB2 ④
user-agent: Python-httpplib2/$Rev: 259 $

,

reply: 'HTTP/1.1 200 OK' ⑤
>>> resp.status
200

```

1. “Cancella questo messaggio di stato.”
2. “Mi dispiace, David, purtroppo non posso farlo.”
3. “Non autorizzato? Hmmph. Cancella questo messaggio di stato, *per favore...*
4. ...e qui ci sono il mio nome utente e la mia password.”
5. “Consideralo fatto!”

E proprio così, puff, è sparito.



14.8. LETTURE DI APPROFONDIMENTO

`httplib2`:

- La pagina del progetto [`httplib2`](#)
- Ulteriori esempi di codice per [`httplib2`](#)
- Usare la cache HTTP nel modo giusto: una introduzione ad [`httplib2`](#)
- [`httplib2`](#): persistenza e autenticazione in HTTP

L'uso della cache in HTTP:

- Tutorial sull'uso della cache in HTTP di Mark Nottingham
- Come controllare la cache con le intestazioni HTTP su Google Doctype

RFC:

- RFC 2616: HTTP
- RFC 2617: la Basic Authentication di HTTP
- RFC 1951: la compressione deflate
- RFC 1952: la compressione gzip

CAPITOLO 15. CASO DI STUDIO: CONVERTIRE chardet VERSO PYTHON 3

“ Parole, non abbiamo altro che parole per andare avanti. ”

— Rosencrantz e Guildenstern sono morti

15.1. IMMERSIONE!

Si deve principalmente a una codifica di caratteri scorretta o sconosciuta la presenza di testo inintelligibile sul web, nella vostra casella di posta e in effetti attraverso ogni sistema computerizzato mai realizzato. Nel capitolo Stringhe ho parlato della storia delle codifiche di carattere e della creazione di Unicode, “una codifica per dominarle tutte”. Mi piacerebbe non dover mai più vedere un carattere incomprensibile su una pagina web, perché tutti i sistemi d’autore memorizzerebbero informazioni di codifica accurate, tutti i protocolli di trasferimento conoscerebbero Unicode e ogni sistema che gestisce testo manterrebbe una fedeltà perfetta nella conversione tra diverse codifiche.

Mi piacerebbe anche avere un pony.

Un pony Unicode.

Uno Unipony, per così dire.

Mi accontenterò del riconoscimento automatico delle codifiche di carattere.



15.2. CHE COS'È IL RICONOSCIMENTO AUTOMATICO DELLE CODIFICHE DI CARATTERE?

Significa prendere una sequenza di byte in una codifica di carattere sconosciuta e cercare di determinare la codifica in modo da poter leggere il testo. È come decifrare un codice quando non avete la chiave di cifratura.

15.2.1. MA NON È IMPOSSIBILE?

In generale, sì. Comunque, alcune codifiche sono ottimizzate per lingue specifiche, e le lingue non sono casuali. Alcune sequenze di caratteri compaiono tutte le volte, mentre altre sequenze non hanno senso. Una persona capace di parlare l'inglese correntemente che apre un giornale e vi trova “txzqJv 2!dasd0a QqdKjvz” riconoscerà istantaneamente che quello non è inglese (sebbene sia composto interamente da lettere inglesi). Analizzando grandi quantità di testo “tipico”, un algoritmo per computer può simulare questa capacità e formulare un'ipotesi fondata sulla lingua in cui è scritto un testo.

In altre parole, il riconoscimento di una codifica è in realtà il riconoscimento di una lingua combinato con la conoscenza di quali lingue tendono a usare determinate codifiche di carattere.

15.2.2. ESISTE UN ALGORITMO DI QUESTO TIPO?

A quanto pare, sì. Tutti browser più diffusi usano il riconoscimento automatico delle codifiche di carattere, perché il web è pieno di pagine che non hanno alcuna informazione sulla loro codifica. Mozilla Firefox contiene una libreria per il riconoscimento automatico delle codifiche distribuita sotto licenza open source. Io ho convertito questa libreria verso Python 2 e ho chiamato il modulo `chardet`. Questo capitolo vi illustrerà passo per passo il processo di conversione del modulo `chardet` da Python 2 verso Python 3.



15.3. UNA INTRODUZIONE AL MODULO `chardet`

Prima di cominciare la conversione, potrebbe essere d'aiuto capire come funziona il codice! Questa sezione vuole essere una breve guida alla struttura e al funzionamento del modulo `chardet`. I sorgenti della libreria sono troppo grandi per riportarne l'intero contenuto qui di seguito, ma potete [scaricarli dal sito `chardet.feedparser.org`](http://chardet.feedparser.org).

Il punto d'ingresso principale per l'algoritmo di riconoscimento è `universaldetector.py`, dove viene definita la classe `UniversalDetector`. (Potreste pensare che il punto d'entrata principale sia la funzione `detect()` in `chardet/__init__.py`, ma in realtà quella è solo una funzione di convenienza che crea un oggetto di tipo `UniversalDetector`, lo invoca e ne restituisce il risultato.)

Ci sono 5 categorie di codifiche che `UniversalDetector` gestisce.

1. UTF-N con BOM. Questa categoria comprende UTF-8, sia le varianti Big-Endian che Little-Endian di UTF-16, e tutte e 4 le varianti di UTF-32 con ordine di byte differente.
2. Codifiche con escape, che sono interamente compatibili con la codifica ASCII a 7 bit, dove i caratteri non-ASCII cominciano con una sequenza di escape. Esempi: ISO-2022-JP (giapponese) e HZ-GB-2312 (cinese).
3. Codifiche multibyte, dove ogni carattere è rappresentato da un numero variabile di byte. Esempi: BIG5 (cinese), SHIFT_JIS (giapponese), EUC-KR (coreano) e UTF-8 senza BOM.
4. Codifiche a singolo byte, dove ogni carattere è rappresentato da un byte. Esempi: KOI8-R (russo), WINDOWS-1255 (ebraico) e TIS-620 (thai).
5. WINDOWS-1252, che viene usato principalmente su Microsoft Windows da direttori d'azienda che non distinguerebbero una codifica di carattere da un buco in terra.

*Il
riconoscimento
di codifica è
in realtà il
riconoscimento
di una lingua
travestito.*

15.3.1. UTF-N CON BOM

Se il testo comincia con un BOM, possiamo ragionevolmente presumere che sia codificato in UTF-8, UTF-16, o UTF-32. (Il BOM ci dirà esattamente quale; questo è quello a cui serve.) Questa categoria viene gestita direttamente da `UniversalDetector`, che restituisce il risultato immediatamente senza ulteriori elaborazioni.

15.3.2. CODIFICHE CON ESCAPE

Se il testo contiene una sequenza di escape riconoscibile che potrebbe indicare una codifica con escape, `UniversalDetector` crea un oggetto `EscCharSetProber` (definito in `escprober.py`) e gli passa il testo.

`EscCharSetProber` crea una serie di macchine a stati finiti, basate sui modelli di HZ-GB-2312, ISO-2022-CN, ISO-2022-JP e ISO-2022-KR (definiti in `escsm.py`). `EscCharSetProber` passa il testo a ognuna di queste macchine a stati, un byte alla volta. Se una qualsiasi di queste macchine a stati riesce a identificare univocamente la codifica, `EscCharSetProber` restituisce immediatamente il risultato positivo a `UniversalDetector`, che lo restituisce al chiamante. Se una qualsiasi delle macchine a stati trova una sequenza illegale, viene scartata e l'elaborazione continua con le altre macchine a stati.

15.3.3. CODIFICHE MULTIBYTE

Presumendo che non ci sia un BOM, `UniversalDetector` controlla se il testo contiene un carattere *high-bit*, cioè un carattere codificato con un byte il cui bit più significativo sia impostato a 1. Se è così, crea una serie di “sonde” per riconoscere le codifiche multibyte, a singolo byte e, come ultima risorsa, WINDOWS-1252.

La sonda per le codifiche multibyte, `MBCSGroupProber` (definita in `mbcsgroupprober.py`), in realtà si occupa solo di gestire un gruppo di altre sonde, una per ogni codifica multibyte: BIG5, GB2312, EUC-TW, EUC-KR, EUC-JP, SHIFT_JIS e UTF-8. `MBCSGroupProber` passa il testo a ognuna di queste specifiche sonde e controlla i risultati. Se una sonda riferisce di aver trovato una sequenza illegale di byte, viene scartata da ulteriori elaborazioni (in modo che, per esempio, ogni chiamata successiva a `UniversalDetector.feed()` eviterà di utilizzare quella sonda). Se una sonda riferisce di essere ragionevolmente sicura di aver riconosciuto la codifica, `MBCSGroupProber` restituisce questo risultato positivo a `UniversalDetector`, che restituisce a sua volta il risultato al chiamante.

La maggior parte delle sonde per le codifiche multibyte eredita da `MultiByteCharSetProber` (definita in `mbcharsetprober.py`) e si occupa semplicemente di agganciare la macchina a stati e l'analizzatore di distribuzione appropriati, lasciando che sia `MultiByteCharSetProber` a fare il resto del lavoro.

`MultiByteCharSetProber` fa scorrere il testo attraverso la macchina a stati della singola codifica, un byte alla volta, per cercare sequenze di byte che indicherebbero un risultato conclusivo positivo o negativo. Allo stesso tempo, `MultiByteCharSetProber` passa il testo a un analizzatore di distribuzione specifico per quella codifica.

Gli analizzatori di distribuzione (tutti definiti in `chardistribution.py`) si basano su modelli, specifici per ogni lingua, che descrivono quali caratteri vengono usati più frequentemente in una lingua. Una volta che `MultiByteCharSetProber` ha passato abbastanza testo all'analizzatore di distribuzione, questo calcola una stima di confidenza basata sul numero di caratteri frequentemente usati, sul numero totale di caratteri e su un rapporto di distribuzione specifico per la lingua. Se la confidenza è abbastanza alta, `MultiByteCharSetProber` restituisce il risultato a `MBCSGroupProber`, che lo restituisce a `UniversalDetector`, che lo restituisce al chiamante.

Il caso del giapponese è più difficile. Gli analizzatori di distribuzione a singolo carattere non sono sempre sufficienti per distinguere tra `EUC-JP` e `SHIFT_JIS`, quindi la classe `SJISProber` (definita in `sjisprober.py`) sfrutta anche l'analisi di distribuzione a 2 caratteri. `SJISContextAnalysis` ed `EUCJPContextAnalysis` (entrambe definite in `jpcntx.py` ed entrambe estensioni della classe `JapaneseContextAnalysis`) controllano la frequenza dei caratteri provenienti dal sillabario Hiragana all'interno del testo. Una volta che è stato elaborato testo a sufficienza, restituiscono un livello di confidenza a `SJISProber`, che controlla il risultato di entrambi gli analizzatori e restituisce il livello di confidenza più alto a `MBCSGroupProber`.

15.3.4. CODIFICHE A SINGOLO BYTE

La sonda per le codifiche a singolo byte, `SBCSGroupProber` (definita in `sbcsgroupprober.py`), si occupa anch'essa di gestire un gruppo di altre sonde, una per ogni combinazione di lingua e codifica a singolo byte: `WINDOWS-1251`, `KOI8-R`, `ISO-8859-5`, `MACCYRILLIC`, `IBM855` e `IBM866` (russo); `ISO-8859-7` e `WINDOWS-1253` (greco); `ISO-8859-5` e `WINDOWS-1251` (bulgaro); `ISO-8859-2` e `WINDOWS-1250` (ungherese); `TIS-620` (thai); `WINDOWS-1255` e `ISO-8859-8` (ebraico).

`SBCSGroupProber` passa il testo a queste sonde, specifiche per coppie di lingua e codifica, e controlla i risultati.

Queste sonde sono tutte implementate come una singola classe, `SingleByteCharSetProber` (definita in

`sbcharsetprober.py`), che prende un modello di lingua come argomento. Il modello di lingua definisce la frequenza con cui differenti sequenze di 2 caratteri appaiono nel testo tipico di quella lingua.

`SingleByteCharSetProber` elabora il testo in ingresso e conta le sequenze di 2 caratteri più utilizzate. Una volta che ha elaborato testo a sufficienza, calcola un livello di confidenza basato sul numero delle sequenze più utilizzate, sul numero totale di caratteri e su un rapporto di distribuzione specifico per lingua.

L'ebraico viene trattato come un caso a parte. Se il testo sembra essere ebraico sulla base dell'analisi di distribuzione a 2 caratteri, la sonda `HebrewProber` (definita in `hebrewprober.py`) prova a distinguere tra ebraico visuale (dove il testo sorgente viene in realtà memorizzato "all'indietro" riga per riga e poi visualizzato così com'è in modo da poter essere letto da destra verso sinistra) ed ebraico logico (dove il testo sorgente viene memorizzato nell'ordine di lettura e poi riportato da destra verso sinistra dal programma che lo visualizza). Dato che alcuni caratteri sono codificati in maniera differente a seconda che compaiano nel mezzo o alla fine di una parola, possiamo fare un'ipotesi ragionevole a proposito della direzione del testo sorgente e restituire la codifica appropriata (`WINDOWS-1255` per l'ebraico logico oppure `ISO-8859-8` per l'ebraico visuale).

*Seramente,
dov'è il mio
pony
Unicode?*

15.3.5. WINDOWS-1252

Se `UniversalDetector` riconosce un carattere high-bit nel testo, ma nessuna delle altre sonde per codifiche multibyte o a singolo byte restituisce un risultato di confidenza, allora crea una sonda `Latin1Prober` (definita in `latin1prober.py`) per cercare di riconoscere testo inglese in una codifica `WINDOWS-1252`. Questo riconoscimento è intrinsecamente inaffidabile, perché le lettere inglesi sono rappresentate allo stesso modo in molte codifiche differenti. L'unico modo di distinguere `WINDOWS-1252` è attraverso simboli comunemente usati come virgolette e apostrofi tipografici, simboli di copyright e simili. `Latin1Prober` riduce automaticamente la propria stima di confidenza per permettere a sonde più accurate di prevalere nel caso sia possibile.



15.4. ESEGUIRE 2to3

Ora effettueremo la migrazione del modulo `chardet` da Python 2 a Python 3. Python 3 viene distribuito con uno script di utilità chiamato `2to3`, che prende in ingresso il vostro codice sorgente in Python 2 e lo converte automaticamente verso Python 3 tanto quanto gli è possibile. In alcuni casi le modifiche sono semplici — una funzione è stata rinominata oppure spostata in un modulo differente — ma in altri casi possono diventare piuttosto complesse. Per avere un'idea di tutto quello che lo script *può* fare, fate riferimento all'appendice Convertire codice verso Python 3 con 2to3. In questo capitolo, cominceremo col lanciare `2to3` sul pacchetto `chardet`, ma come vedrete ci sarà ancora molto lavoro da fare dopo che gli strumenti automatici avranno eseguito i loro incantesimi.

Il pacchetto `chardet` è suddiviso in molti file diversi, tutti nella stessa directory. Lo script `2to3` facilita la conversione di più file in una volta: basta passargli una directory come argomento dalla riga di comando e `2to3` convertirà ognuno dei file a turno.

```
C:\home\chardet> python c:\Python30\Tools\Scripts\2to3.py -w chardet\
```

```
RefactoringTool: Salto il correttore implicito: buffer
```

```
RefactoringTool: Salto il correttore implicito: idioms
```

```
RefactoringTool: Salto il correttore implicito: set_literal
```

```
RefactoringTool: Salto il correttore implicito: ws_comma
```

```
--- chardet\__init__.py (originale)
```

```
+++ chardet\__init__.py (modificato)
```

```
@@ -18,7 +18,7 @@
```

```
__version__ = "1.0.1"
```

```
def detect(aBuf):
```

```
- import universaldetector
```

```
+ from . import universaldetector
```

```
    u = universaldetector.UniversalDetector()
```

```
    u.reset()
```

```
    u.feed(aBuf)
```

```
--- chardet\big5prober.py (originale)
```

```
+++ chardet\big5prober.py (modificato)
```

```
@@ -25,10 +25,10 @@
```

```
# 02110-1301 USA
```

```
##### END LICENSE BLOCK #####
```

```
-from mbcharsetprober import MultiByteCharSetProber
```

```
-from codingstatemachine import CodingStateMachine
```

```
-from chardistribution import Big5DistributionAnalysis
```

```
-from mbcssm import Big5SMMModel
```

```
+from .mbcharsetprober import MultiByteCharSetProber
```

```
+from .codingstatemachine import CodingStateMachine
```

```
+from .chardistribution import Big5DistributionAnalysis
```

```
+from .mbcssm import Big5SMMModel
```

```
class Big5Prober(MultiByteCharSetProber):
```

```
    def __init__(self):
```

```
--- chardet\chardistribution.py (originale)
```

```
+++ chardet\chardistribution.py (modificato)
```

```
@@ -25,12 +25,12 @@
```

```
# 02110-1301 USA
```

```
##### END LICENSE BLOCK #####
```

```
-import constants
```

```
-from euctwfreq import EUCTWCharToFreqOrder, EUCTW_TABLE_SIZE, EUCTW_TYPICAL_DISTRIBUTION_RATIO
```

```
-from euckrfreq import EUCKRCharToFreqOrder, EUCKR_TABLE_SIZE, EUCKR_TYPICAL_DISTRIBUTION_RATIO
```

```
-from gb2312freq import GB2312CharToFreqOrder, GB2312_TABLE_SIZE, GB2312_TYPICAL_DISTRIBUTION_RATIO
```

```
-from big5freq import Big5CharToFreqOrder, BIG5_TABLE_SIZE, BIG5_TYPICAL_DISTRIBUTION_RATIO
```

```
-from jisfreq import JISCharToFreqOrder, JIS_TABLE_SIZE, JIS_TYPICAL_DISTRIBUTION_RATIO
```

```
+from . import constants
```

```
+from .euctwfreq import EUCTWCharToFreqOrder, EUCTW_TABLE_SIZE, EUCTW_TYPICAL_DISTRIBUTION_RATIO
```

```
+from .euckrfreq import EUCKRCharToFreqOrder, EUCKR_TABLE_SIZE, EUCKR_TYPICAL_DISTRIBUTION_RATIO
```

```
+from .gb2312freq import GB2312CharToFreqOrder, GB2312_TABLE_SIZE, GB2312_TYPICAL_DISTRIBUTION_RATIO
```

```
+from .big5freq import Big5CharToFreqOrder, BIG5_TABLE_SIZE, BIG5_TYPICAL_DISTRIBUTION_RATIO
```

```
+from .jisfreq import JISCharToFreqOrder, JIS_TABLE_SIZE, JIS_TYPICAL_DISTRIBUTION_RATIO
```

```
ENOUGH_DATA_THRESHOLD = 1024
```

```
SURE_YES = 0.99
```

```
.
```

```
.
```

```
. (va avanti così per un po')
```

```
.
```

```
.
```

```
RefactoringTool: File che sono stati modificati:
```

```
RefactoringTool: chardet\__init__.py
```

```
RefactoringTool: chardet\big5prober.py
```

```
RefactoringTool: chardet\chardistribution.py
```

```
RefactoringTool: chardet\charsetgroupprober.py
```

```
RefactoringTool: chardet\codingstatemachine.py
```

```
RefactoringTool: chardet\constants.py
```

```
RefactoringTool: chardet\escprober.py
```

```
RefactoringTool: chardet\escsm.py
```

RefactoringTool: chardet\eucjpprober.py
RefactoringTool: chardet\euuckrprober.py
RefactoringTool: chardet\euectwprober.py
RefactoringTool: chardet\gb2312prober.py
RefactoringTool: chardet\hebrewprober.py
RefactoringTool: chardet\jpcntx.py
RefactoringTool: chardet\langbulgarianmodel.py
RefactoringTool: chardet\langcyrillicmodel.py
RefactoringTool: chardet\langgreekmodel.py
RefactoringTool: chardet\langhebrewmodel.py
RefactoringTool: chardet\langhungarianmodel.py
RefactoringTool: chardet\langthaimodel.py
RefactoringTool: chardet\latin1prober.py
RefactoringTool: chardet\mbcharsetprober.py
RefactoringTool: chardet\mbcsgroupprober.py
RefactoringTool: chardet\mbcssm.py
RefactoringTool: chardet\sbcharsetprober.py
RefactoringTool: chardet\sbcsgroupprober.py
RefactoringTool: chardet\sjisprober.py
RefactoringTool: chardet\universaldetector.py
RefactoringTool: chardet=utf8prober.py

Ora eseguiamo lo script 2to3 sul programma di collaudo, test.py.

```
C:\home\chardet> python c:\Python30\Tools\Scripts\2to3.py -w test.py
```

```
RefactoringTool: Salto il correttore implicito: buffer
```

```
RefactoringTool: Salto il correttore implicito: idioms
```

```
RefactoringTool: Salto il correttore implicito: set_literal
```

```
RefactoringTool: Salto il correttore implicito: ws_comma
```

```
--- test.py (originale)
```

```
+++ test.py (modificato)
```

```
@@ -4,7 +4,7 @@
```

```
count = 0
```

```
u = UniversalDetector()
```

```
for f in glob.glob(sys.argv[1]):
```

```
- print f.ljust(60),
```

```
+ print(f.ljust(60), end=' ')
```

```
    u.reset()
```

```
    for line in file(f, 'rb'):
```

```
        u.feed(line)
```

```
@@ -12,8 +12,8 @@
```

```
    u.close()
```

```
    result = u.result
```

```
    if result['encoding']:
```

```
-         print result['encoding'], 'con confidenza', result['confidence']
```

```
+         print(result['encoding'], 'con confidenza', result['confidence'])
```

```
    else:
```

```
-         print '***** nessun risultato'
```

```
+         print('***** nessun risultato')
```

```
    count += 1
```

```
-print count, 'test'
```

```
+print(count, 'test')
```

```
RefactoringTool: File che sono stati modificati:
```

```
RefactoringTool: test.py
```


Be', non è stato così difficile. Ci sono state solo alcune istruzioni print e import da convertire. Parlando di queste ultime, qual era il problema con tutte quelle istruzioni di importazione? Per rispondere a questa domanda, avete bisogno di capire come il modulo chardet è diviso in molteplici file.



15.5. UNA BREVE DIGRESSIONE SUI MODULI MULTIFILE

`chardet` è un *modulo multifile*. Avrei potuto scegliere di mettere tutto il codice in un unico file (chiamato `chardet.py`), ma non l'ho fatto. Invece, ho creato una directory (chiamata `chardet`) e poi ho creato un file `__init__.py` in quella directory. Se Python vede un file `__init__.py` in una directory, assume che tutti i file in quella directory siano parte dello stesso modulo. Il nome del modulo è il nome della directory. I file nella directory possono fare riferimento ad altri file nella stessa directory, o persino all'interno di sottodirectory. (Ne parlerò più in dettaglio fra un minuto.) Ma l'intera collezione di file viene presentata ad altro codice Python come un singolo modulo — come se tutte le funzioni e le classi fossero in un singolo file `.py`.

Che cosa va messo in un file `__init__.py`? Niente. Tutto. Qualcosa. Il file `__init__.py` non ha bisogno di definire nulla, può letteralmente essere un file vuoto. Oppure potete usarlo per definire le funzioni dei vostri punti d'ingresso principali. O potete metterci tutte le vostre funzioni. O tutte tranne una.

 Una directory contenente un file `__init__.py` viene sempre trattata come un modulo multifile. Senza un file `__init__.py`, una directory è semplicemente una directory contenente file `.py` non correlati tra loro.

Vediamo come questi moduli funzionano in pratica.

```
>>> import chardet

>>> dir(chardet) ①
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__path__', '__version__', 'detect']

>>> chardet ②
<module 'chardet' from 'C:\Python31\lib\site-packages\chardet\__init__.py'>
```

I. A parte i soliti attributi di classe, l'unica cosa contenuta nel modulo `chardet` è la funzione `detect()`.

2. Ecco il vostro primo indizio che il modulo `chardet` sia più di un semplice file: il “modulo” viene mostrato come il file `__init__.py` nella directory `chardet`.

Diamo uno sguardo a quel file `__init__.py`.

```
def detect(aBuf):                                ①
    from . import universaldetector              ②
    u = universaldetector.UniversalDetector()
    u.reset()
    u.feed(aBuf)
    u.close()
    return u.result
```

1. Il file `__init__.py` definisce la funzione `detect()`, che è il punto d'ingresso principale per la libreria `chardet`.
2. Ma la funzione `detect()` contiene pochissimo codice! In effetti, tutto quello che fa è importare il modulo `universaldetector` e cominciare a usarlo. Ma dov'è definito `universaldetector`?

La risposta si trova in quella strana istruzione `import`:

```
from . import universaldetector
```

Tradotta in italiano, quella istruzione significa “importa il modulo `universaldetector` che si trova nella stessa directory in cui sono io”, dove “io” è il file `chardet/__init__.py`. Questa viene chiamata *importazione relativa* ed è il modo in cui i file all'interno di un modulo multifile possono fare riferimento gli uni agli altri, senza preoccuparsi di eventuali conflitti di nomi con gli altri moduli che potreste avere installato nel vostro percorso di ricerca per le importazioni. Questa istruzione `import` cercherà il modulo `universaldetector` solo all'interno della directory `chardet`.

Questi due concetti — `__init__.py` e le importazioni relative — significano che potete suddividere il vostro modulo in tutte le parti che preferite. Il modulo `chardet` comprende 36 file `.py` — 36! Eppure tutto quello che avete bisogno di fare per cominciare a usarlo è scrivere `import chardet`, poi potete chiamare la funzione principale `chardet.detect()`. All'insaputa del vostro codice, la funzione `detect()` è in realtà definita nel file `chardet/__init__.py`. E a vostra insaputa, la funzione `detect()` usa un'importazione relativa per fare

riferimento a una classe definita in `chardet/universaldetector.py`, che a sua volta usa importazioni relative per altri cinque file, tutti contenuti nella directory `chardet/`.

☞ Se vi siete mai trovati a scrivere una libreria Python di grandi dimensioni (o, più probabilmente, quando realizzate che le dimensioni della vostra piccola libreria sono cresciute fino a diventare grandi), prendetevi il tempo di riorganizzarla in un modulo multifile. Questa è una delle molte cose che Python è capace di fare bene, quindi avvantaggiatevene.

*
**

15.6. RISOLVERE QUELLO CHE 2to3 NON PUÒ

15.6.1. False È SINTASSI NON VALIDA

E ora, la verifica vera e propria: lanciamo il programma di collaudo con i nostri test. Dato che i test sono stati progettati per coprire tutti i possibili percorsi di esecuzione nel codice, questo è un buon modo di collaudare il codice convertito per assicurarsi che non ci sia alcun bug in agguato da qualche parte.

```
C:\home\chardet> python test.py tests\**
```

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

```
File "test.py", line 1, in <module>
```

```
    from chardet.universaldetector import UniversalDetector
```

```
File "C:\home\chardet\chardet\universaldetector.py", line 51
```

```
    self.done = constants.False
```

```
    ^
```

```
SyntaxError: invalid syntax
```

*Avete i test,
giusto?*

Hmm, un piccolo intoppo. In Python 3, `False` è una parola riservata e quindi non potete usarla come nome di variabile. Diamo un'occhiata a `constants.py` per vedere dov'è definita. Ecco la versione originale della parte rilevante di `constants.py`, prima che lo script `2to3` la modificasse:

```
import __builtin__

if not hasattr(__builtin__, 'False'):
    False = 0
    True = 1
else:
    False = __builtin__.False
    True = __builtin__.True
```

Questo frammento di codice è progettato per permettere a questa libreria di venire eseguita dalle versioni più vecchie di Python 2. Prima della versione 2.3, Python non aveva alcun tipo `bool`. Questo codice riconosce l'assenza delle costanti built-in `True` e `False` e le definisce nel caso sia necessario.

Comunque, Python 3 avrà sempre un tipo `bool`, quindi questo intero frammento di codice è superfluo. La soluzione più semplice consiste nel sostituire tutte le istanze di `constants.True` e `constants.False` con `True` e `False` rispettivamente e poi cancellare questo codice ormai inutile da `constants.py`.

Così questa riga in `universaldetector.py`:

```
self.done = constants.False
```

Diventa:

```
self.done = False
```

Aaah, non siete soddisfatti? Il codice è più corto e già più leggibile.

15.6.2. NESSUN MODULO CHIAMATO `constants`

È il momento di eseguire nuovamente `test.py` e vedere fino a dove riesce ad arrivare.

```
C:\home\chardet> python test.py tests\*\*
```

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

```
File "test.py", line 1, in <module>
```

```
    from chardet.universaldetector import UniversalDetector
```

```
File "C:\home\chardet\chardet\universaldetector.py", line 29, in <module>
```

```
    import constants, sys
```

```
ImportError: No module named constants
```

Cosa dice? Nessun modulo chiamato constants? Ma certo che c'è un modulo chiamato constants. È proprio lì, in `chardet/constants.py`.

Ricordate quando lo script `2to3` ha corretto tutte quelle istruzioni `import`? Questa libreria contiene un sacco di `import` relativi — cioè, moduli che importano altri moduli nell'ambito della libreria — ma *la logica dietro il funzionamento delle importazioni relative è cambiata in Python 3*. In Python 2, potevate semplicemente scrivere `import constants` e l'interprete avrebbe guardato nella directory `chardet` come prima cosa. In Python 3, tutte le istruzioni `import` sono assolute per default. Se volete importare un modulo in maniera relativa in Python 3, dovete farlo esplicitamente:

```
from . import constants
```

Ma aspettate. Non doveva essere lo script `2to3` a curarsi di queste cose per voi? Be', lo ha fatto, ma la particolare istruzione `import` che ha generato l'errore combina due differenti tipi di importazione in una sola riga: un'importazione relativa del modulo `constants` all'interno della libreria e un'importazione assoluta del modulo `sys` che è preinstallato nella libreria standard di Python. In Python 2, potevate combinare questi due tipi in un'unica istruzione `import`. In Python 3 non potete, ma lo script `2to3` non è abbastanza scaltro da dividere l'istruzione `import` in due.

La soluzione consiste nel dividere l'istruzione `import` manualmente. Così questa doppia istruzione `import`:

```
import constants, sys
```

Deve diventare due istruzioni `import` separate:

```
from . import constants
import sys
```

Ci sono variazioni di questo problema sparse in tutta la libreria `chardet`. In alcuni posti è “`import constants, sys`”; in altri posti è “`import constants, re`”. La soluzione è la stessa: dividete manualmente l’istruzione `import` in due righe, una per quella relativa, l’altra per quella assoluta.

Procediamo!

15.6.3. IL NOME 'file' NON È DEFINITO

Ed eccoci ancora qui, a lanciare `test.py` per provare a eseguire i nostri test...

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    for line in file(f, 'rb'):
NameError: name 'file' is not defined
```

Questo errore mi ha sorpreso, perché ho usato questo idioma fin da quando riesco a ricordare. In Python 2, la funzione globale `file()` era un alias per `open()`, che era la funzione standard da usare per aprire i file e leggerli. In Python 3, la funzione globale `file()` non esiste più, ma la funzione `open()` esiste ancora.

*open() è il
nuovo file().
PapayaWhip
è il nuovo
nero.*

Quindi, la soluzione più semplice al problema della mancanza di `file()` è chiamare `open()` al suo posto:

```
for line in open(f, 'rb'):
```

E questo è tutto quello che ho da dire al riguardo.

15.6.4. IMPOSSIBILE USARE UN PATTERN DI STRINGA SU UN OGGETTO TIPO `bytes`

Ora le cose cominciano a diventare interessanti. E per “interessanti” voglio dire “incasinate come l’inferno”.

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 98, in feed
    if self._highBitDetector.search(aBuf):
TypeError: can't use a string pattern on a bytes-like object
```

Per correggere questo errore, diamo un’occhiata a che cos’è `self._highBitDetector`. È definito nel metodo `__init__()` della classe `UniversalDetector`:

```
class UniversalDetector:
    def __init__(self):
        self._highBitDetector = re.compile(r'[\x80-\xFF]')
```

Questo codice precompila un’espressione regolare progettata per trovare caratteri di tipo non-ASCII nell’intervallo 128–255 (0x80–0xFF). Aspettate, non è proprio esatto; devo essere più preciso nella mia terminologia. Questo pattern è utilizzato per trovare *byte* di tipo non-ASCII nell’intervallo 128–255.

E il problema è proprio qui.

In Python 2, una stringa era un array di `byte` la cui codifica di carattere veniva memorizzata separatamente. Se volevate che Python 2 tenesse traccia della codifica di carattere, dovevate usare una stringa di tipo Unicode (`u''`). Ma in Python 3 una stringa è sempre ciò che Python 2 chiamava una stringa di tipo Unicode — cioè un array di caratteri Unicode (eventualmente con lunghezze diverse in termini di `byte`). Dato che questa espressione regolare è definita come un pattern di stringa, può essere usata solo per fare ricerche in una stringa — ancora, un array di caratteri. Ma ciò in cui stiamo cercando non è una stringa, bensì un array di `byte`. La *traceback*, cioè la traccia dello stack di esecuzione, ci dice che questo errore è apparso in `universaldetector.py`:

```
def feed(self, aBuf):
    .
    .
    .
    if self._mInputState == ePureAscii:
        if self._highBitDetector.search(aBuf):
```

E che cos'è aBuf? Torniamo ancora più indietro al punto in cui `UniversalDetector.feed()` viene invocata. Quel punto si trova nel programma di collaudo, `test.py`.

```
u = UniversalDetector()
.
.
.
for line in open(f, 'rb'):
    u.feed(line)
```

E qui troviamo la nostra risposta: nel metodo `UniversalDetector.feed()`, `aBuf` è una riga letta da un file su disco. Guardate attentamente i parametri utilizzati per aprire il file: `'rb'`. `'r'` sta per “read”, lettura; va bene, grazie tante, stiamo leggendo il file. Ah, ma `'b'` sta per “binary”, binario. Senza il flag `'b'`, questo ciclo `for` leggerebbe il file riga per riga e convertirebbe ogni riga in una stringa — un array di caratteri Unicode — in accordo con la codifica di carattere predefinita del sistema. Ma con il flag `'b'` questo ciclo `for` legge il file riga per riga e memorizza ogni riga esattamente come appare nel file, sotto forma di un array di byte. Quell'array di byte viene passato a `UniversalDetector.feed()` e alla fine viene passato all'espressione regolare precompilata `self._highBitDetector` per cercare i... caratteri high-bit. Ma non abbiamo caratteri. Abbiamo byte. Oops.

*Non un array
di caratteri,
ma un array
di byte.*

Quello su cui abbiamo bisogno che l'espressione regolare effettui la ricerca non è un array di caratteri, ma un array di byte.

Una volta che realizzate questo, la soluzione non è difficile. Le espressioni regolari definite come stringhe possono effettuare ricerche sulle stringhe. Le espressioni regolari definite come array di byte possono effettuare ricerche sugli array di byte. Per definire un pattern come un array di byte, cambiamo semplicemente il tipo dell'argomento che usiamo per definire l'espressione regolare ad array di byte. (C'è un altro caso di questo stesso problema, proprio nella riga successiva.)

```
class UniversalDetector:
    def __init__(self):
-         self._highBitDetector = re.compile(r'[\x80-\xFF]')
-         self._escDetector = re.compile(r'(\033|~{})')
+         self._highBitDetector = re.compile(b'[\x80-\xFF]')
+         self._escDetector = re.compile(b'(\033|~{})')
        self._mEscCharSetProber = None
        self._mCharSetProbers = []
        self.reset()
```

Una ricerca sull'intera base di codice rivela altri due utilizzi del modulo `re`, in `charsetprober.py`. Ancora, il codice definisce espressioni regolari come stringhe ma le utilizza su `aBuf` che è un array di byte. La soluzione è la stessa: definire i pattern di espressione regolare come array di byte.

```

class CharSetProber:
    .
    .
    .
    def filter_high_bit_only(self, aBuf):
-         aBuf = re.sub(r'([\x00-\x7F])+', ' ', aBuf)
+         aBuf = re.sub(b'([\x00-\x7F])+', b' ', aBuf)
        return aBuf

    def filter_without_english_letters(self, aBuf):
-         aBuf = re.sub(r'([A-Za-z])+', ' ', aBuf)
+         aBuf = re.sub(b'([A-Za-z])+', b' ', aBuf)
        return aBuf

```

15.6.5. IMPOSSIBILE CONVERTIRE IMPLICITAMENTE UN OGGETTO 'bytes' IN str

Sempre più stranissimo...

```

C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 100, in feed
    elif (self._mInputState == ePureAscii) and self._escDetector.search(self._mLastChar + aBuf):
TypeError: Can't convert 'bytes' object to str implicitly

```

Qui c'è una sfortunata collisione tra stile di codifica e interprete Python. L'errore di tipo `TypeError` potrebbe essere ovunque su quella riga, ma la traceback non vi dice esattamente dov'è. Potrebbe essere nella prima condizione o nella seconda, ma la traceback sarebbe la stessa. Per circoscrivere l'errore dovete spezzare la riga a metà in questo modo:

```

elif (self._mInputState == ePureAscii) and \
    self._escDetector.search(self._mLastChar + aBuf):

```


E rieseguire il test:

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 101, in feed
    self._escDetector.search(self._mLastChar + aBuf):
TypeError: Can't convert 'bytes' object to str implicitly
```

Aha! Il problema non era nella prima condizione (`self._mInputState == ePureAscii`) ma nella seconda. Quindi cosa potrebbe causare un `TypeError` in quel punto? Forse state pensando che il metodo `search()` si aspetti un valore di tipo differente, ma quel problema non genererebbe questa traceback. Le funzioni Python possono accettare qualsiasi valore; se passate il numero corretto di argomenti, la funzione verrà eseguita. Potrebbe *fallire* se le passate un valore di tipo differente rispetto a quello che si aspetta, ma se questo fosse successo la traceback punterebbe da qualche parte all'interno della funzione. Invece questa traceback dice che l'interprete non è mai arrivato tanto lontano da chiamare il metodo `search()`. Quindi il problema deve essere in quella operazione `+`, dato che sta cercando di costruire un valore che alla fine verrà passato al metodo `search()`.

Sappiamo da una correzione precedente che `aBuf` è un array di byte. Quindi cos'è `self._mLastChar`? È una variabile di istanza, definita nel metodo `reset()`, che viene effettivamente chiamato dal metodo `__init__()`.

```

class UniversalDetector:
    def __init__(self):
        self._highBitDetector = re.compile(b'[\x80-\xFF]')
        self._escDetector = re.compile(b'(\033|~{)')
        self._mEscCharSetProber = None
        self._mCharSetProbers = []

        self.reset()

    def reset(self):
        self.result = {'encoding': None, 'confidence': 0.0}
        self.done = False
        self._mStart = True
        self._mGotData = False
        self._mInputState = ePureAscii

        self._mLastChar = ''

```

E ora abbiamo la nostra risposta. La vedete? `self._mLastChar` è una stringa, ma `aBuf` è un array di byte. E non potete concatenare una stringa a un array di byte — nemmeno una stringa di lunghezza zero.

Quindi cos'è `self._mLastChar` ad ogni modo? Guardiamo nel metodo `feed()`, giusto qualche riga più in basso rispetto a dove la traceback si è generata.

```

if self._mInputState == ePureAscii:
    if self._highBitDetector.search(aBuf):
        self._mInputState = eHighbyte
    elif (self._mInputState == ePureAscii) and \
        self._escDetector.search(self._mLastChar + aBuf):
        self._mInputState = eEscAscii

self._mLastChar = aBuf[-1]

```

La funzione chiamante invoca continuamente questo metodo `feed()` con pochi byte alla volta. Il metodo elabora i byte che gli sono stati dati (passati come `aBuf`), poi memorizza l'ultimo byte in `self._mLastChar` nel caso ce ne sia bisogno durante l'invocazione successiva. (In una codifica multibyte, il metodo `feed()` potrebbe

venire invocato con metà di un carattere e poi invocato ancora con l'altra metà.) Ma dato che ora `aBuf` è un array di byte invece di essere una stringa, anche `self._mLastChar` deve essere un array di byte. Perciò:

```
def reset(self):  
    .  
    .  
    .  
-    self._mLastChar = ''  
+    self._mLastChar = b''
```

Una ricerca su tutta la base di codice per “`mLastChar`” rivela un problema simile in `mbcharsetprober.py`, dove invece di memorizzare l'ultimo carattere vengono memorizzati gli ultimi *due* caratteri. La classe `MultiByteCharSetProber` usa una lista di stringhe di 1 carattere per tenere traccia degli ultimi due caratteri; in Python 3 deve usare una lista di interi, perché non sta realmente tenendo traccia di caratteri, ma di byte. (I byte sono semplicemente interi nell'intervallo 0–255.)

```
class MultiByteCharSetProber(CharSetProber):  
    def __init__(self):  
        CharSetProber.__init__(self)  
        self._mDistributionAnalyzer = None  
        self._mCodingSM = None  
-        self._mLastChar = ['\x00', '\x00']  
+        self._mLastChar = [0, 0]  
  
    def reset(self):  
        CharSetProber.reset(self)  
        if self._mCodingSM:  
            self._mCodingSM.reset()  
        if self._mDistributionAnalyzer:  
            self._mDistributionAnalyzer.reset()  
-        self._mLastChar = ['\x00', '\x00']  
+        self._mLastChar = [0, 0]
```

15.6.6. TIPI DI OPERANDO NON SUPPORTATI PER +: 'int' E 'bytes'

Ho una buona notizia e una cattiva notizia. La buona notizia è che stiamo facendo passi avanti...

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 101, in feed
    self._escDetector.search(self._mLastChar + aBuf):
TypeError: unsupported operand type(s) for +: 'int' and 'bytes'
```

...La cattiva notizia è che questo non sempre sembra un passo avanti.

Ma questo è un passo avanti! Davvero! Anche se la traceback evidenzia la stessa riga di codice, questo è un errore diverso da quello di prima. Un passo avanti! Quindi qual è ora il problema? L'ultima volta che ho controllato, questa riga di codice non provava a concatenare un int con un array di byte (bytes). Infatti, avete appena speso un sacco di tempo per assicurarvi che self._mLastChar fosse un array di byte. Come si è trasformato in un int?

La risposta non è nelle righe di codice precedenti, ma nelle righe seguenti.

```
if self._mInputState == ePureAscii:
    if self._highBitDetector.search(aBuf):
        self._mInputState = eHighbyte
    elif (self._mInputState == ePureAscii) and \
        self._escDetector.search(self._mLastChar + aBuf):
        self._mInputState = eEscAscii

self._mLastChar = aBuf[-1]
```

Questo errore non avviene la prima volta che il metodo `feed()` viene chiamato; avviene la *seconda volta*, dopo che a `self._mLastChar` è stato assegnato l'ultimo byte di `aBuf`. E quindi, qual è il problema con quella operazione? Recuperare un singolo elemento da un array di byte produce un intero, non un array di byte. Per vedere la differenza, seguitemi nella shell interattiva:

```
>>> aBuf = b'\xEF\xBB\xBF' ①
>>> len(aBuf)
3
>>> mLastChar = aBuf[-1]
>>> mLastChar ②
191
>>> type(mLastChar) ③
<class 'int'>
>>> mLastChar + aBuf ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

*Ogni
elemento in
una stringa è
una stringa.
Ogni
elemento in
un array di
byte è un
intero.*

TypeError: unsupported operand type(s) for +: 'int' and 'bytes'

```
>>> mLastChar = aBuf[-1:] ⑤
>>> mLastChar
b'\xbf'
>>> mLastChar + aBuf ⑥
b'\xbf\xef\xbb\xbf'
```

1. Definisce un array di byte di lunghezza 3.
2. L'ultimo elemento dell'array di byte è 191.

3. Quell'ultimo elemento è un intero.
4. Concatenare un intero a un array di byte non funziona. Avete appena riprodotto l'errore che è stato trovato in `universaldetector.py`.
5. Ah, questa è la soluzione. Invece di prendere l'ultimo elemento dell'array di byte, affettate la lista per creare un nuovo array di byte contenente solo l'ultimo elemento. Cioè, cominciate con l'ultimo elemento e prolungate la fetta fino alla fine dell'array di byte. Ora `mLastChar` è un array di byte di lunghezza 1.
6. Concatenare un array di byte di lunghezza 1 con un array di byte di lunghezza 3 restituisce un nuovo array di byte di lunghezza 4.

Quindi, per assicurarvi che il metodo `feed()` in `universaldetector.py` continui a lavorare a prescindere da quante volte viene invocato, dovete inizializzare `self._mLastChar` come un array di byte di lunghezza 0 e poi *assicurarvi che rimanga un array di byte*.

```
self._escDetector.search(self._mLastChar + aBuf):  
self._mInputState = eEscAscii
```

```
- self._mLastChar = aBuf[-1]  
+ self._mLastChar = aBuf[-1:]
```

15.6.7. `ord()` SI ASPETTA UNA STRINGA DI LUNGHEZZA 1, MA TROVA UN `int`

Già stanchi? Ci siete quasi...

```

C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml          ascii con confidenza 1.0
tests\Big5\0804.blogspot.com.xml

Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 116, in feed
    if prober.feed(aBuf) == constants.eFoundIt:
  File "C:\home\chardet\chardet\charsetgroupprober.py", line 60, in feed
    st = prober.feed(aBuf)
  File "C:\home\chardet\chardet\utf8prober.py", line 53, in feed
    codingState = self._mCodingSM.next_state(c)
  File "C:\home\chardet\chardet\codingstatemachine.py", line 43, in next_state
    byteCls = self._mModel['classTable'][ord(c)]
TypeError: ord() expected string of length 1, but int found

```

Bene, quindi `c` è un `int` ma la funzione `ord()` si aspetta una stringa di 1 carattere. Abbastanza onesto. Dove viene definito `c`?

```

# codingstatemachine.py
def next_state(self, c):
    # recuperiamo la classe di ogni byte
    # se è il primo byte, ne recuperiamo anche la lunghezza
    byteCls = self._mModel['classTable'][ord(c)]

```

Questo frammento di codice non è di aiuto; il parametro viene semplicemente passato dentro la funzione. Risaliamo la traccia dello stack di esecuzione.

```

# utf8prober.py
def feed(self, aBuf):
    for c in aBuf:
        codingState = self._mCodingSM.next_state(c)

```

Vedete? In Python 2 `aBuf` era una stringa, quindi `c` era una stringa di 1 carattere. (Questo è quello che si ottiene quando si itera su una stringa — tutti i caratteri, uno a uno.) Ma ora `aBuf` è un vettore di byte, quindi `c` è un `int`, non una stringa di 1 carattere. In altre parole, non c'è alcun bisogno di chiamare la funzione `ord()` perché `c` è già un `int`!

Perciò:

```
def next_state(self, c):
    # recuperiamo la classe di ogni byte
    # se è il primo byte, ne recuperiamo anche la lunghezza
-   byteCls = self._mModel['classTable'][ord(c)]
+   byteCls = self._mModel['classTable'][c]
```

Una ricerca sull'intera base di codice per istanze di “`ord(c)`” rivela problemi simili in `sbcharsetprober.py`...

```
# sbcharsetprober.py
def feed(self, aBuf):
    if not self._mModel['keepEnglishLetter']:
        aBuf = self.filter_without_english_letters(aBuf)
    aLen = len(aBuf)
    if not aLen:
        return self.get_state()
    for c in aBuf:
        order = self._mModel['charToOrderMap'][ord(c)]
```

...e in `latin1prober.py`...

```
# latin1prober.py
def feed(self, aBuf):
    aBuf = self.filter_with_english_letters(aBuf)
    for c in aBuf:
        charClass = Latin1_CharToClass[ord(c)]
```


c sta iterando su aBuf, il che significa che è un intero, non una stringa di 1 carattere. La soluzione è la stessa: sostituire ord(c) con il solo c.

```
# sbcharsetprober.py
def feed(self, aBuf):
    if not self._mModel['keepEnglishLetter']:
        aBuf = self.filter_without_english_letters(aBuf)
    aLen = len(aBuf)
    if not aLen:
        return self.get_state()
    for c in aBuf:
-         order = self._mModel['charToOrderMap'][ord(c)]
+         order = self._mModel['charToOrderMap'][c]
```

```
# latin1prober.py
def feed(self, aBuf):
    aBuf = self.filter_with_english_letters(aBuf)
    for c in aBuf:
-         charClass = Latin1_CharToClass[ord(c)]
+         charClass = Latin1_CharToClass[c]
```

15.6.8. TIPI NON ORDINABILI: int() >= str()

Continuiamo ancora.

```

C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml          ascii con confidenza 1.0
tests\Big5\0804.blogspot.com.xml

Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 116, in feed
    if prober.feed(aBuf) == constants.eFoundIt:
  File "C:\home\chardet\chardet\charsetgroupprober.py", line 60, in feed
    st = prober.feed(aBuf)
  File "C:\home\chardet\chardet\sjisprober.py", line 68, in feed
    self._mContextAnalyzer.feed(self._mLastChar[2 - charLen :], charLen)
  File "C:\home\chardet\chardet\jpcntx.py", line 145, in feed
    order, charLen = self.get_order(aBuf[i:i+2])
  File "C:\home\chardet\chardet\jpcntx.py", line 176, in get_order
    if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
TypeError: unorderable types: int() >= str()

```

Quindi qual è l'errore in questo caso? “Unorderable types”? Tipi non ordinabili? Ancora una volta, la differenza tra array di byte e stringhe sta rialzando la sua ripugnante testa. Date un'occhiata al codice:

```

class SJISContextAnalysis(JapaneseContextAnalysis):
    def get_order(self, aStr):
        if not aStr: return -1, 1
        # trova la lunghezza in byte del carattere corrente
        if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
            ((aStr[0] >= '\xE0') and (aStr[0] <= '\xFC')):
            charLen = 2
        else:
            charLen = 1

```

E da dove viene aStr? Risaliamo la traccia dello stack di esecuzione:

```
def feed(self, aBuf, aLen):
    .
    .
    .
    i = self._mNeedToSkipCharNum
    while i < aLen:
        order, charLen = self.get_order(aBuf[i:i+2])
```

Oh, guardate, è il nostro vecchio amico `aBuf`. Come avrete potuto intuire da tutti gli altri problemi che abbiamo incontrato in questo capitolo, `aBuf` è un array di byte. Qui il metodo `feed()` non sta semplicemente passandolo tutto intero, ma lo sta affettando. Però, come avete visto precedentemente in questo capitolo, affettare un array di byte restituisce un array di byte, quindi il parametro `aStr` che viene passato al metodo `get_order()` è ancora un array di byte.

E cos'è che questo codice sta cercando di fare con `aStr`? Prende il primo elemento dell'array di byte e lo confronta con una stringa di lunghezza 1. In Python 2 questo funzionava, perché `aStr` e `aBuf` erano stringhe, e `aStr[0]` sarebbe stata una stringa, e potevate confrontare le stringhe per la disuguaglianza. Ma in Python 3 `aStr` e `aBuf` sono array di byte, `aStr[0]` è un intero e non potete effettuare un confronto di disuguaglianza tra interi e stringhe senza convertire esplicitamente gli uni nel tipo delle altre o viceversa.

In questo caso non c'è bisogno di rendere il codice più complicato aggiungendo una conversione esplicita. `aStr[0]` produce un intero e le cose con le quali state effettuando il confronto sono tutte costanti. Modifichiamole da stringhe di 1 carattere a interi. E visto che ci siamo, cambiamo il nome di `aStr` in `aBuf`, dato che in realtà non è una stringa.

```

class SJISContextAnalysis(JapaneseContextAnalysis):
-     def get_order(self, aStr):
-         if not aStr: return -1, 1
+     def get_order(self, aBuf):
+         if not aBuf: return -1, 1

        # trova la lunghezza in byte del carattere corrente
-         if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
-             ((aStr[0] >= '\xE0') and (aStr[0] <= '\xFC')):
+         if ((aBuf[0] >= 0x81) and (aBuf[0] <= 0x9F)) or \
+             ((aBuf[0] >= 0xE0) and (aBuf[0] <= 0xFC)):

            charLen = 2
        else:
            charLen = 1

        # restituisce il suo ordinale se è hiragana
-         if len(aStr) > 1:
-             if (aStr[0] == '\202') and \
-                 (aStr[1] >= '\x9F') and \
-                 (aStr[1] <= '\xF1'):
-                 return ord(aStr[1]) - 0x9F, charLen
+         if len(aBuf) > 1:
+             if (aBuf[0] == 202) and \
+                 (aBuf[1] >= 0x9F) and \
+                 (aBuf[1] <= 0xF1):
+                 return aBuf[1] - 0x9F, charLen

        return -1, charLen

```

```

class EUCJPContextAnalysis(JapaneseContextAnalysis):
-     def get_order(self, aStr):
-         if not aStr: return -1, 1
+     def get_order(self, aBuf):
+         if not aBuf: return -1, 1

        # trova la lunghezza in byte del carattere corrente

```

```

-         if (aStr[0] == '\x8E') or \
-             ((aStr[0] >= '\xA1') and (aStr[0] <= '\xFE')):
+         if (aBuf[0] == 0x8E) or \
+             ((aBuf[0] >= 0xA1) and (aBuf[0] <= 0xFE)):
            charLen = 2
-         elif aStr[0] == '\x8F':
+         elif aBuf[0] == 0x8F:
            charLen = 3
        else:
            charLen = 1

```

restituisce il suo ordinale se è hiragana

```

-         if len(aStr) > 1:
-             if (aStr[0] == '\xA4') and \
-                 (aStr[1] >= '\xA1') and \
-                 (aStr[1] <= '\xF3'):
-                 return ord(aStr[1]) - 0xA1, charLen
+         if len(aBuf) > 1:
+             if (aBuf[0] == 0xA4) and \
+                 (aBuf[1] >= 0xA1) and \
+                 (aBuf[1] <= 0xF3):
+                 return aBuf[1] - 0xA1, charLen

```

```

return -1, charLen

```

Una ricerca sull'intera base di codice per occorrenze della funzione `ord()` rivela lo stesso problema in `chardistribution.py` (più precisamente, nelle classi `EUCTWDistributionAnalysis`, `EUCKRDistributionAnalysis`, `GB2312DistributionAnalysis`, `Big5DistributionAnalysis`, `SJISDistributionAnalysis` e `EUCJPDistributionAnalysis`). In ogni caso, la correzione è simile alla modifica che abbiamo fatto alle classi `EUCJPContextAnalysis` e `SJISContextAnalysis` in `jpcntx.py`.

15.6.9. IL NOME GLOBALE 'reduce' NON È DEFINITO

Ancora una volta sulla breccia...

```

C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml          ascii con confidenza 1.0
tests\Big5\0804.blogspot.com.xml

Traceback (most recent call last):
  File "test.py", line 12, in <module>
    u.close()
  File "C:\home\chardet\chardet\universaldetector.py", line 141, in close
    proberConfidence = prober.get_confidence()
  File "C:\home\chardet\chardet\latin1prober.py", line 126, in get_confidence
    total = reduce(operator.add, self._mFreqCounter)
NameError: global name 'reduce' is not defined

```

Secondo [la guida ufficiale alle novità di Python 3](#), la funzione `reduce()` è stata spostata dallo spazio di nomi globale al modulo `functools`. Per citare la guida: “Usate `functools.reduce()` se proprio ne avete bisogno; comunque, il 99 per cento delle volte un esplicito ciclo `for` è più leggibile.” Potete approfondire le motivazioni di questa decisione sul blog di Guido van Rossum: [Il destino di `reduce\(\)` in Python 3000](#).

```

def get_confidence(self):
    if self.get_state() == constants.eNotMe:
        return 0.01

```

```

total = reduce(operator.add, self._mFreqCounter)

```

La funzione `reduce()` prende due argomenti — una funzione e una lista (strettamente parlando, qualsiasi oggetto iterabile potrebbe andare) — e applica una funzione cumulativamente a ogni elemento della lista. In altre parole, questo è un modo elaborato e tortuoso di sommare tutti gli elementi di una lista e restituire il risultato.

Questa mostruosità era così comune che a Python è stata aggiunta una funzione globale `sum()`.

```
def get_confidence(self):  
    if self.get_state() == constants.eNotMe:  
        return 0.01
```

```
- total = reduce(operator.add, self._mFreqCounter)  
+ total = sum(self._mFreqCounter)
```

Dato che non state più usando il modulo `operator`, potete anche rimuovere quella istruzione `import` dall'inizio del file.

```
from .charsetprober import CharSetProber  
from . import constants  
- import operator
```

I CAN HAZ TESTZ?

```
C:\home\chardet> python test.py tests\**
```

tests\ascii\howto.diveintomark.org.xml	ascii con confidenza 1.0
tests\Big5\0804.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\blog.worren.net.xml	Big5 con confidenza 0.99
tests\Big5\carbonxiv.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\catshadow.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\coolloud.org.tw.xml	Big5 con confidenza 0.99
tests\Big5\digitalwall.com.xml	Big5 con confidenza 0.99
tests\Big5\ebao.us.xml	Big5 con confidenza 0.99
tests\Big5\fudesign.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\kafkatseng.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\ke207.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\leavesth.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\letterlego.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\linyijen.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\marilynwu.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\myblog.pchome.com.tw.xml	Big5 con confidenza 0.99
tests\Big5\oui-design.com.xml	Big5 con confidenza 0.99
tests\Big5\sanwenji.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\sinica.edu.tw.xml	Big5 con confidenza 0.99
tests\Big5\sylvia1976.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\tlkkuo.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\tw.blog.xubg.com.xml	Big5 con confidenza 0.99
tests\Big5\unoriginalblog.com.xml	Big5 con confidenza 0.99
tests\Big5\upsaid.com.xml	Big5 con confidenza 0.99
tests\Big5\willythecop.blogspot.com.xml	Big5 con confidenza 0.99
tests\Big5\ytc.blogspot.com.xml	Big5 con confidenza 0.99
tests\EUC-JP\aiivy.co.jp.xml	EUC-JP con confidenza 0.99
tests\EUC-JP\akaname.main.jp.xml	EUC-JP con confidenza 0.99
tests\EUC-JP\arclamp.jp.xml	EUC-JP con confidenza 0.99

.

.

.

```
316 test
```


Porca puttana, funziona davvero! *Ime fa una piccola danza*



15.7. RIEPILOGO

Che cosa abbiamo imparato?

1. Convertire una qualunque quantità non banale di codice da Python 2 verso Python 3 sarà doloroso. Non c'è alcun modo di evitarlo. È difficile.
2. Lo strumento automatico 2to3 è utile per quello che può fare, ma si occuperà solo della parte facile — rinominerà le funzioni e i moduli, modificherà la sintassi. È un impressionante esempio di ingegneria, ma in fondo non è altro che un automa intelligente per la ricerca e la sostituzione.
3. Il problema n°1 nella conversione di questa libreria era la differenza tra stringhe e byte. In questo caso sembrava ovvio, dato che lo scopo stesso della libreria chardet è convertire un flusso di byte in una stringa. Ma “un flusso di byte” scorre molto più spesso di quanto possiate immaginare. Leggete un file in modalità “binaria”? Avrete un flusso di byte. Prelevate una pagina web? Chiamate una API web? Anche quelle restituiscono un flusso di byte.
4. Voi avete bisogno di capire il vostro programma. A fondo. Preferibilmente perché lo avete scritto, ma è necessario almeno che vi troviate a vostro agio con tutte le sue stranezze e i suoi angoli ammuffiti. I bug si annidano ovunque.
5. I test sono essenziali. Non convertite nulla senza di loro. L'unica ragione per cui ho una qualche confidenza che chardet funzioni in Python 3 è perché ho cominciato con una serie di test che esercitava tutti i percorsi principali all'interno del codice. Se non avete nessun test, scrivetene alcuni prima di cominciare a convertire verso Python 3. Se avete alcuni test, scrivetene altri. Se avete molti test, allora il vero divertimento può cominciare.

CAPITOLO 16. DISTRIBUIRE LIBRERIE PYTHON

“ Scoprirai che la vergogna è come il dolore; si sente solo una volta. ”

— Marchesa di Merteuil, Le relazioni pericolose

16.1. IMMERSIONE!

Tra l'artista e la sua consacrazione c'è di mezzo la consegna dell'opera finita, o almeno questo è quello che sostengono alcuni. Se volete rilasciare un programma, una libreria, un framework, un'applicazione scritta in Python, potete utilizzare il framework di distribuzione Distutils incluso in Python 3. Distutils è molte cose: uno strumento di assemblaggio (per voi), uno strumento di installazione (per i vostri utenti), un formato di metadati per pacchetti software (per i motori di ricerca) e anche di più. Si integra con il Python Package Index (“PyPI”, Indice dei Pacchetti Python), un archivio centrale per librerie Python open source.

Tutte queste caratteristiche di Distutils si basano su un *programma di installazione* tradizionalmente chiamato `setup.py`. In effetti, avete già visto diversi programmi di installazione di Distutils in questo libro. Avete usato Distutils per installare `httplib2` nel capitolo Servizi web HTTP e ancora per installare `chardet` nel capitolo Caso di studio: convertire chardet verso Python 3.

In questo capitolo, imparerete come funzionano i programmi di installazione per `chardet` e `httplib2` e seguirete passo per passo il processo di rilascio del vostro software Python.

```
# file setup.py di chardet
from distutils.core import setup
setup(
    name = "chardet",
    packages = ["chardet"],
    version = "1.0.2",
    description = "Riconoscitore di codifiche universale",
    author = "Mark Pilgrim",
    author_email = "mark@diveintomark.org",
    url = "http://chardet.feedparser.org/",
    download_url = "http://chardet.feedparser.org/download/python3-chardet-1.0.1.tgz",
    keywords = ["encoding", "i18n", "xml"],
    classifiers = [
        "Programming Language :: Python",
        "Programming Language :: Python :: 3",
        "Development Status :: 4 - Beta",
        "Environment :: Other Environment",
        "Intended Audience :: Developers",
        "License :: OSI Approved :: GNU Library or Lesser General Public License (LGPL)",
        "Operating System :: OS Independent",
        "Topic :: Software Development :: Libraries :: Python Modules",
        "Topic :: Text Processing :: Linguistic",
    ],
    long_description = """\
```

Riconoscitore di codifiche universale

Riconosce

- ASCII, UTF-8, UTF-16 (2 varianti), UTF-32 (4 varianti)
- Big5, GB2312, EUC-TW, HZ-GB-2312, ISO-2022-CN (Cinese tradizionale e semplificato)
- EUC-JP, SHIFT_JIS, ISO-2022-JP (Giapponese)
- EUC-KR, ISO-2022-KR (Coreano)
- KOI8-R, MacCyrillic, IBM855, IBM866, ISO-8859-5, windows-1251 (Cirillico)
- ISO-8859-2, windows-1250 (Ungherese)

- ISO-8859-5, windows-1251 (Bulgaro)
- windows-1252 (Inglese)
- ISO-8859-7, windows-1253 (Greco)
- ISO-8859-8, windows-1255 (Ebraico visuale e logico)
- TIS-620 (Thai)

Questa versione richiede Python 3 o successivo; una versione per Python 2 è disponibile separatamente.

```
"""
```

```
)
```

👉 `chardet` e `httplib2` sono open source, ma non c'è alcun requisito che vi obbliga a rilasciare le vostre librerie Python secondo i termini di qualche licenza particolare. Il processo descritto in questo capitolo funzionerà per qualsiasi software Python a prescindere dalla licenza.

*
**

16.2. COSE CHE DISTUTILS NON PUÒ FARE PER VOI

Rilasciare il vostro primo pacchetto Python è un processo che potrebbe intimidirvi. (Rilasciare il secondo è un po' più facile.) Distutils cerca di automatizzare questo processo per quanto è possibile, ma ci sono alcune cose che dovete proprio fare da soli.

- **Scegliete una licenza.** Questo è un argomento complicato, gravido di politica e di problemi. Se desiderate rilasciare il vostro software come open source, vi offro umilmente cinque consigli:
 1. Non scrivete la vostra licenza.
 2. Non scrivete la vostra licenza.
 3. Non scrivete la vostra licenza.
 4. La licenza non deve per forza essere GPL, ma deve essere GPL-compatibile.
 5. Non scrivete la vostra licenza.

- **Classificate il vostro software** usando il sistema di classificazione offerto da PyPI. Spiegherò cosa significa questo più avanti in questo capitolo.
- **Scrivete un file “leggimi”**. Non risparmiatevi su questo. Come minimo, il file dovrebbe offrire ai vostri utenti una descrizione di quello che fa il vostro software e di come installarlo.



16.3. LA STRUTTURA DELLE DIRECTORY

Per cominciare a impacchettare il vostro software, avete bisogno di mettere ordine tra i vostri file e le vostre directory. La directory di `httplib2` è strutturata in questo modo:

```

httplib2/           ①
|
+--README.txt       ②
|
+--setup.py         ③
|
+--httplib2/        ④
|
+--__init__.py
|
+--iri2uri.py
```

1. Create una directory radice per contenere ogni cosa. Datele lo stesso nome del vostro modulo Python.
2. Per favorire gli utenti Windows, il vostro file “leggimi” dovrebbe includere un’estensione `.txt` e dovrebbe usare i ritorni a capo in stile Windows. Solo perché *voi* usate un esotico editor di testo che lanciate dalla riga di comando e che include il proprio linguaggio di macro non significa che dovete rendere la vita difficile ai vostri utenti. (I vostri utenti usano Blocco note. Triste ma vero.) Anche se vi trovate su Linux o Mac OS X, il vostro esotico editor di testo senza dubbio è dotato di un’opzione per salvare i file con i ritorni a capo in stile Windows.

3. Il vostro programma di installazione per Distutils dovrebbe essere chiamato `setup.py` a meno che non abbiate una buona ragione per chiamarlo diversamente. E non avete una buona ragione per chiamarlo diversamente.
4. Se il vostro software Python è composto da un singolo file `.py`, dovrete mettere quel file nella directory radice a fianco del vostro file “leggi” e del vostro programma di installazione. Ma `httplib2` non è composto da un singolo file `.py`, bensì è un modulo multifile. Ma questo va benissimo! Semplicemente, mettete la directory `httplib2` nella directory radice, in modo da avere un file `__init__.py` all’interno della directory `httplib2/` all’interno della directory radice `httplib2/`. Questo non è un problema, e in effetti semplificherà il vostro processo di impacchettamento.

La directory `chardet` è strutturata in maniera leggermente differente. Come `httplib2`, anche `chardet` è un modulo multifile, quindi contiene una directory `chardet/` all’interno della directory radice `chardet/`. In aggiunta al file `README.txt`, `chardet` include la propria documentazione formattata in HTML nella directory `docs/`. La directory `docs/` contiene diversi file `.html` e `.css` e una sottodirectory `images/` che contiene diversi file `.png` e `.gif`. (Questo risulterà importante più avanti.) Infine, per rispettare le convenzioni del software rilasciato sotto licenza (L)GPL, include un file separato chiamato `COPYING.txt` che contiene il testo completo della licenza LGPL.

```
chardet/
|
+--COPYING.txt
|
+--setup.py
|
+--README.txt
|
+--docs/
|  |
|  +--index.html
|  |
|  +--usage.html
|  |
|  +--images/ ...
|
+--chardet/
|
+--__init__.py
|
+--big5freq.py
|
+--...
```

*
**

16.4. SCRIVERE IL VOSTRO PROGRAMMA DI INSTALLAZIONE

Il programma di installazione di Distutils è un programma Python. In teoria, può fare qualsiasi cosa che Python è in grado di fare. In pratica, dovrebbe limitarsi a fare il meno possibile nel modo più standard possibile. I programmi di installazione dovrebbero essere noiosi. Più esotico è il vostro processo di installazione, più esotici saranno i bug che vi verranno segnalati.

La prima riga di tutti i programmi di installazione di Distutils è sempre la stessa:

```
from distutils.core import setup
```

Questa riga importa la funzione `setup()`, che è il punto d'ingresso principale di Distutils. Il 95% di tutti i programmi di installazione di Distutils è composto da una singola invocazione a `setup()` e basta. (Mi sono assolutamente appena inventato questa statistica, ma se il vostro programma di installazione di Distutils fa altre cose oltre a invocare la funzione `setup()`, dovrete avere una buona ragione. Avete una buona ragione? Non penso proprio.)


La funzione `setup()` può accettare dozzine di parametri. Per salvaguardare la sanità mentale di chiunque sia coinvolto, dovete usare gli argomenti con nome per tutti i parametri. Questa non è semplicemente una convenzione, ma un requisito vincolante. Il vostro programma di installazione si bloccherà se provate a invocare la funzione `setup()` con argomenti senza nome.

I seguenti argomenti con nome sono obbligatori:

- **name**, il nome del pacchetto;
- **version**, il numero di versione del pacchetto;
- **author**, il vostro nome completo;
- **author_email**, il vostro indirizzo email;
- **url**, la pagina web principale del vostro progetto, che può essere la pagina del pacchetto ospitata da PyPI se non avete un sito web separato per il progetto.

Sebbene non siano richiesti, vi raccomando di includere anche i seguenti argomenti nel vostro programma di installazione:

- **description**, una riga di riepilogo del progetto;
- **long_description**, una stringa su più righe in formato reStructuredText, che PyPI converte in HTML e mostra sulla pagina del vostro pacchetto;
- **classifiers**, una lista di stringhe formattate in modo speciale che vengono descritte nella prossima sezione.

 I metadati da utilizzare nei programmi di installazione sono definiti nella PEP 314.

Ora diamo un'occhiata al programma di installazione per `chardet`. Contiene tutti questi parametri obbligatori e raccomandati, più uno che non ho ancora menzionato: `packages`.

```
from distutils.core import setup
setup(
    name = 'chardet',
    packages = ['chardet'],
    version = '1.0.2',
    description = 'Riconoscitore di codifiche universale',
    author = 'Mark Pilgrim',
    ...
)
```

Il parametro `packages` evidenzia una sfortunata sovrapposizione di vocaboli nel processo di distribuzione. Abbiamo parlato del “pacchetto” come della cosa che state assemblando (e potenzialmente elencando nell'Indice dei Pacchetti Python). Il parametro `packages` non si riferisce a questo, ma al fatto che il modulo `chardet` è un modulo multifile, talvolta noto come... “pacchetto”. Il parametro `packages` dice a Distutils di includere la directory `chardet/`, il suo file `__init__.py` e tutti gli altri file `.py` che costituiscono il modulo `chardet`. Questo è piuttosto importante: tutte le nostre allegre chiacchiere su documentazione e metadati sono irrilevanti se dimenticate di includere il codice vero e proprio!



16.5. CLASSIFICARE IL VOSTRO PACCHETTO

Il Python Package Index (“PyPI”, Indice dei Pacchetti Python) contiene migliaia di librerie Python. I corretti metadati di classificazione consentiranno alle persone interessate di trovare la vostra libreria più facilmente. PyPI vi consente di scorrere i pacchetti per classificatore. Potete anche selezionare molteplici classificatori per restringere il campo della vostra ricerca. I classificatori non sono metadati invisibili che potete semplicemente ignorare!

Per classificare il vostro software, passate un parametro `classifiers` alla funzione `setup()` di Distutils. Il parametro `classifiers` è una lista di stringhe. Il formato di queste stringhe *non* è libero, perché tutte le stringhe di classificazione dovrebbero provenire da questa lista su PyPI.

I classificatori sono opzionali. Potete scrivere un programma di installazione di Distutils senza alcun classificatore. **Non fatelo.** Dovreste *sempre* includere almeno i seguenti classificatori.

- **Programming Language.** In particolare, dovrete includere sia "Programming Language :: Python" che "Programming Language :: Python :: 3". Se non li includete, il vostro pacchetto non verrà mostrato in questa lista di librerie compatibili con Python 3, che viene segnalata a lato di ogni singola pagina del sito `pypi.python.org`.
- **License.** Questa è *la prima cosa in assoluto che controllo* quando sto considerando librerie di terze parti. Non obbligatemi a dare la caccia a questa informazione vitale. Non includete più di un classificatore di licenza a meno che il vostro software non sia esplicitamente rilasciato secondo i termini di più di una licenza. (E non rilasciate software secondo i termini di più di una licenza a meno che non siate obbligati a farlo. E non obbligate altri sviluppatori a farlo. La questione delle licenze è già una seccatura così com'è, non peggioratela.)
- **Operating System.** Se il vostro software funziona solo su Windows (o Mac OS X, o Linux), voglio saperlo prima, non dopo. Se il vostro software funziona ovunque senza alcun codice specifico per la piattaforma, usate il classificatore "Operating System :: OS Independent". Classificatori Operating System molteplici sono necessari solo se il vostro software richiede uno specifico supporto per ogni piattaforma. (Questo non è molto comune.)

Vi raccomando di includere anche i seguenti classificatori.

- **Development Status.** Il vostro software ha la qualità di una versione beta? Alfa? Pre-alfa? Sceglietene una. Siate onesti.
- **Intended Audience.** Chi scaricherebbe il vostro software? Le scelte più comuni sono Developers, End Users/Desktop, Science/Research e System Administrators.
- **Framework.** Se il vostro software è un componente aggiuntivo per un framework Python più grande come Django o Zope, includete l'appropriato classificatore Framework, altrimenti omettetelo.
- **Topic.** C'è un vasta gamma di argomenti tra cui scegliere; scegliete tutti quelli che si applicano al vostro caso.

16.5.1. ESEMPI DI BUONI CLASSIFICATORI

Per fare un esempio, ecco i classificatori di Django, un framework per applicazioni web da eseguire sul vostro server web che è pronto per essere usato in ambienti di produzione, indipendente dalla piattaforma e distribuito secondo i termini della licenza BSD. (Django non è ancora compatibile con Python 3, quindi il classificatore Programming Language :: Python :: 3 non viene elencato.)

```
Programming Language :: Python
License :: OSI Approved :: BSD License
Operating System :: OS Independent
Development Status :: 5 - Production/Stable
Environment :: Web Environment
Framework :: Django
Intended Audience :: Developers
Topic :: Internet :: WWW/HTTP
Topic :: Internet :: WWW/HTTP :: Dynamic Content
Topic :: Internet :: WWW/HTTP :: WSGI
Topic :: Software Development :: Libraries :: Python Modules
```

Ecco i classificatori di chardet, la libreria per il riconoscimento delle codifiche di carattere descritta nel capitolo Caso di studio: convertire chardet verso Python 3. chardet è di qualità beta, indipendente dalla piattaforma, compatibile con Python 3, distribuita secondo la licenza LGPL e pensata per essere integrata nei prodotti di altri sviluppatori.

```
Programming Language :: Python
Programming Language :: Python :: 3
License :: OSI Approved :: GNU Library or Lesser General Public License (LGPL)
Operating System :: OS Independent
Development Status :: 4 - Beta
Environment :: Other Environment
Intended Audience :: Developers
Topic :: Text Processing :: Linguistic
Topic :: Software Development :: Libraries :: Python Modules
```

Ed ecco i classificatori di `httplib2`, la libreria su cui si basa il capitolo `Servizi web HTTP`. `httplib2` è di qualità beta, indipendente dalla piattaforma, distribuita secondo la licenza MIT e rivolta agli sviluppatori che programmano in Python.

```
Programming Language :: Python
Programming Language :: Python :: 3
License :: OSI Approved :: MIT License
Operating System :: OS Independent
Development Status :: 4 - Beta
Environment :: Web Environment
Intended Audience :: Developers
Topic :: Internet :: WWW/HTTP
Topic :: Software Development :: Libraries :: Python Modules
```

16.6. SPECIFICARE FILE AGGIUNTIVI CON UN MANIFESTO

Per default, Distutils includerà i seguenti file nel vostro pacchetto di distribuzione:

- `README.txt`;
- `setup.py`;
- i file `.py` che occorrono ai moduli multifile elencati nel parametro `packages`;
- i singoli file `.py` elencati nel parametro `py_modules`.

Questo includerà tutti i file del progetto `httplib2`. Ma per il progetto `chardet` vogliamo anche includere il file di licenza `COPYING.txt` e l'intera directory `docs/` che contiene immagini e file HTML. Per dire a Distutils di includere anche questi file e queste directory aggiuntive quando assembla il pacchetto di distribuzione per `chardet`, avete bisogno di un *file manifesto*.

Un file manifesto è un file di testo chiamato `MANIFEST.in`. Mettetelo nella directory radice del progetto, a fianco di `README.txt` e `setup.py`. I file manifesto *non* sono programmi Python, ma sono file di testo che contengono una serie di “comandi” in un formato definito da Distutils. I comandi di manifesto vi permettono di includere o escludere specifici file e directory.

Questo è l'intero file manifesto per il progetto `chardet`:

`include COPYING.txt` ①

`recursive-include docs *.html *.css *.png *.gif` ②

1. La prima riga è ovvia: include il file `COPYING.txt` che si trova nella directory radice del progetto.
2. La seconda riga è un po' più complicata. Il comando `recursive-include` prende un nome di directory e uno o più nomi di file. I nomi di file non si limitano a indicare file specifici, ma possono includere wildcard. Questa riga significa: "Vedi quella directory `docs/` nella directory radice del progetto? Cerca (ricorsivamente) in quella directory i file `.html`, `.css`, `.png` e `.gif`. Li voglio tutti nel mio pacchetto di distribuzione."

Tutti i comandi di manifesto preservano la struttura di directory che avete impostato nella directory del vostro progetto. Quel comando `recursive-include` non metterà un gruppo di file `.html` e `.png` nella directory radice del pacchetto di distribuzione. Invece, manterrà la struttura della directory `docs/` esistente, includendo solo quei file contenuti dentro quella directory che corrispondono alle wildcard date. (Non l'ho detto prima, ma la documentazione di `chardet` è in realtà scritta in XML e convertita in HTML da un programma separato. Non voglio includere i file XML nel pacchetto di distribuzione, ma solo i file HTML e le immagini.)



I file manifesto hanno il proprio singolare formato. Leggete [Specificare i file da distribuire](#) e [I comandi per i modelli di manifesto se volete conoscere i dettagli.](#)

Per ripetermi: avete bisogno di creare un file manifesto solo se volete includere file che Distutils non include per default. Se avete bisogno di un file manifesto, dovrete includere solo i file e le directory che Distutils non sarebbe altrimenti in grado di trovare da solo.

16.7. CONTROLLARE CHE IL VOSTRO PROGRAMMA DI INSTALLAZIONE NON CONTENGA ERRORI

Ci sono molte cose di cui tenere traccia. Distutils include un comando di validazione predefinito che controlla la presenza di tutti i metadati obbligatori nel vostro programma di installazione. Per esempio, se dimenticate di includere il parametro `version`, Distutils ve lo ricorderà.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check  
eseguo il comando check  
attenzione: controllo: metadato richiesto mancante: version
```

Una volta che includete un parametro version (e tutti gli altri metadati obbligatori), l'esecuzione del comando check somiglierà a questa:

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check  
eseguo il comando check
```

*
**

16.8. CREARE UNA DISTRIBUZIONE DEI SORGENTI

Distutils supporta l'assemblaggio di diversi tipi di pacchetti di distribuzione. Come minimo, dovrete creare una “distribuzione dei sorgenti” che contenga il vostro codice sorgente, il vostro programma di setup di Distutils, il vostro file “leggimi” e tutti i file aggiuntivi che volete includere. Per creare una distribuzione dei sorgenti passate il comando `sdist` al vostro programma di installazione di Distutils.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py sdist
```

eseguo il comando sdist

eseguo il comando check

leggo il modello di manifesto 'MANIFEST.in'

scrivo il file di manifesto 'MANIFEST'

creo chardet-1.0.2

creo chardet-1.0.2\chardet

creo chardet-1.0.2\docs

creo chardet-1.0.2\docs\images

copio i file in chardet-1.0.2...

copio COPYING -> chardet-1.0.2

copio README.txt -> chardet-1.0.2

copio setup.py -> chardet-1.0.2

copio chardet__init__.py -> chardet-1.0.2\chardet

copio chardet\big5freq.py -> chardet-1.0.2\chardet

...

copio chardet\universaldetector.py -> chardet-1.0.2\chardet

copio chardet\utf8prober.py -> chardet-1.0.2\chardet

copio docs\faq.html -> chardet-1.0.2\docs

copio docs\history.html -> chardet-1.0.2\docs

copio docs\how-it-works.html -> chardet-1.0.2\docs

copio docs\index.html -> chardet-1.0.2\docs

copio docs\license.html -> chardet-1.0.2\docs

copio docs\supported-encodings.html -> chardet-1.0.2\docs

copio docs\usage.html -> chardet-1.0.2\docs

copio docs\images\caution.png -> chardet-1.0.2\docs\images

copio docs\images\important.png -> chardet-1.0.2\docs\images

copio docs\images\note.png -> chardet-1.0.2\docs\images

copio docs\images\permalink.gif -> chardet-1.0.2\docs\images

copio docs\images\tip.png -> chardet-1.0.2\docs\images

copio docs\images\warning.png -> chardet-1.0.2\docs\images

creo 'dist'

creo 'dist\chardet-1.0.2.zip' e vi aggiungo 'chardet-1.0.2'

aggiungo 'chardet-1.0.2\COPYING'

```
aggiungo 'chardet-1.0.2\PKG-INFO'
aggiungo 'chardet-1.0.2\README.txt'
aggiungo 'chardet-1.0.2\setup.py'
aggiungo 'chardet-1.0.2\chardet\big5freq.py'
aggiungo 'chardet-1.0.2\chardet\big5prober.py'
...
aggiungo 'chardet-1.0.2\chardet\universaldetector.py'
aggiungo 'chardet-1.0.2\chardet\utf8prober.py'
aggiungo 'chardet-1.0.2\chardet\__init__.py'
aggiungo 'chardet-1.0.2\docs\faq.html'
aggiungo 'chardet-1.0.2\docs\history.html'
aggiungo 'chardet-1.0.2\docs\how-it-works.html'
aggiungo 'chardet-1.0.2\docs\index.html'
aggiungo 'chardet-1.0.2\docs\license.html'
aggiungo 'chardet-1.0.2\docs\supported-encodings.html'
aggiungo 'chardet-1.0.2\docs\usage.html'
aggiungo 'chardet-1.0.2\docs\images\caution.png'
aggiungo 'chardet-1.0.2\docs\images\important.png'
aggiungo 'chardet-1.0.2\docs\images\note.png'
aggiungo 'chardet-1.0.2\docs\images\permalink.gif'
aggiungo 'chardet-1.0.2\docs\images\tip.png'
aggiungo 'chardet-1.0.2\docs\images\warning.png'
rimuovo 'chardet-1.0.2' (e ogni cosa al suo interno)
```

Ci sono diverse cose da notare qui:

- Distutils ha notato la presenza del file manifesto (MANIFEST.in);
- Distutils ha riconosciuto con successo il contenuto del file manifesto aggiungendo i file che volevamo — COPYING.txt, i file HTML e le immagini contenute nella directory docs/;
- se guardate nella vostra directory di progetto, vedrete che Distutils ha creato una directory dist/ in cui si trova il file .zip che potete distribuire.


```
c:\Users\pilgrim\chardet> dir dist
```

Il volume nell'unità C non ha etichetta.

Numero di serie del volume: DED5-B4F8

Directory di c:\Users\pilgrim\chardet\dist

```
30/07/2009  06:29    <DIR>          .
30/07/2009  06:29    <DIR>          ..
30/07/2009  06:29                206,440 chardet-1.0.2.zip
                1 File                206.440 byte
                2 Directory    61.424.635.904 byte disponibili
```

*
**

16.9. CREARE UN PROGRAMMA DI INSTALLAZIONE GRAFICO

A mio parere, tutte le librerie Python si meritano un programma di installazione grafico per utenti Windows. È facile da creare (anche se voi stessi non usate Windows) e gli utenti Windows lo apprezzano.

Distutils può creare un programma di installazione grafico per Windows se passate il comando `bdist_wininst` al vostro programma di installazione di Distutils.

```

c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py bdist_wininst
eseguo il comando bdist_wininst
eseguo il comando build
eseguo il comando build_py
creo build
creo build\lib
creo build\lib\chardet
copio chardet\big5freq.py -> build\lib\chardet
copio chardet\big5prober.py -> build\lib\chardet
...
copio chardet\universaldetector.py -> build\lib\chardet
copio chardet\utf8prober.py -> build\lib\chardet
copio chardet\__init__.py -> build\lib\chardet
installo in build\bdist.win32\wininst
eseguo il comando install_lib
creo build\bdist.win32
creo build\bdist.win32\wininst
creo build\bdist.win32\wininst\PURELIB
creo build\bdist.win32\wininst\PURELIB\chardet
copio build\lib\chardet\big5freq.py -> build\bdist.win32\wininst\PURELIB\chardet
copio build\lib\chardet\big5prober.py -> build\bdist.win32\wininst\PURELIB\chardet
...
copio build\lib\chardet\universaldetector.py -> build\bdist.win32\wininst\PURELIB\chardet
copio build\lib\chardet\utf8prober.py -> build\bdist.win32\wininst\PURELIB\chardet
copio build\lib\chardet\__init__.py -> build\bdist.win32\wininst\PURELIB\chardet
eseguo il comando install_egg_info
Scrivo build\bdist.win32\wininst\PURELIB\chardet-1.0.2-py3.1.egg-info
creo 'c:\users\pilgrim\appdata\local\temp\tmp2f4h7e.zip' e vi aggiungo '.'
aggiungo 'PURELIB\chardet-1.0.2-py3.1.egg-info'
aggiungo 'PURELIB\chardet\big5freq.py'
aggiungo 'PURELIB\chardet\big5prober.py'
...
aggiungo 'PURELIB\chardet\universaldetector.py'
aggiungo 'PURELIB\chardet\utf8prober.py'

```

```

aggiungo 'PURELIB\chardet\__init__.py'
rimuovo 'build\bdist.win32\wininst' (e ogni cosa al suo interno)
c:\Users\pilgrim\chardet> dir dist

Il volume nell'unità C non ha etichetta.

Numero di serie del volume: DED5-B4F8

```

Directory di c:\Users\pilgrim\chardet\dist

```

30/07/2009  10:14    <DIR>          .
30/07/2009  10:14    <DIR>          ..
30/07/2009  10:14                371,236 chardet-1.0.2.win32.exe
30/07/2009  06:29                206,440 chardet-1.0.2.zip
                2 File                577.676 byte
                2 Directory        61.424.070.656 byte disponibili

```

16.9.1. ASSEMBLARE PACCHETTI DI INSTALLAZIONE PER ALTRI SISTEMI OPERATIVI

Distutils può aiutarvi ad assemblare pacchetti di installazione per gli utenti Linux. A mio parere, probabilmente non ne vale la pena. Se volete che il vostro software venga distribuito per Linux, spendereste meglio il vostro tempo lavorando con i membri della comunità che si sono specializzati nel creare pacchetti software per le distribuzioni Linux più importanti.

Per esempio, la mia libreria chardet si trova negli archivi della distribuzione Debian GNU/Linux (e di conseguenza anche negli archivi di Ubuntu). Io non ho avuto niente a che fare con questo: un giorno il pacchetto è semplicemente comparso. La comunità di Debian ha le proprie politiche per distribuire librerie Python e il pacchetto Debian python-chardet è realizzato in modo da seguire queste convenzioni. Dato che il pacchetto si trova negli archivi della distribuzione Debian, gli utenti Debian riceveranno aggiornamenti di sicurezza e/o nuove versioni a seconda delle impostazioni di sistema con cui hanno deciso di gestire il proprio computer.

I pacchetti Linux che vengono assemblati da Distutils non offrono nessuno di questi vantaggi. Il vostro tempo è speso meglio da qualche altra parte.



16.10. AGGIUNGERE IL VOSTRO SOFTWARE ALL'INDICE DEI PACCHETTI PYTHON

Caricare il vostro software sull'Indice dei Pacchetti Python è un processo che si svolge in tre fasi.

1. Registrazione di un utente
2. Registrazione del software
3. Caricamento del pacchetto creato con `setup.py sdist` e `setup.py bdist_*`

Per registrarvi, andate alla [pagina di registrazione per gli utenti di PyPI](#). Inserite il nome utente e la password che desiderate, fornite un indirizzo email valido e cliccate il bottone Register. (Se avete una chiave PGP o GPG potete fornire anche quella. Se non ne avete una o non sapete cosa significa, non fatelo.) Controllate la vostra casella email: entro pochi minuti dovrete ricevere un messaggio da PyPI contenente un link di verifica. Cliccate sul link per completare il processo di registrazione.

Ora dovete registrare il vostro software su PyPI e caricarlo. Potete farlo in un unico passo.

```

c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py register sdist bdist_wininst upload ①
eseguo il comando register
Abbiamo bisogno di sapere chi sei, quindi scegli una tra le seguenti alternative:
1. usa i tuoi dati di accesso esistenti,
2. registrati come nuovo utente,
3. chiedi al server di generarti una nuova password (e di spedirtela via email), o
4. esci
La tua selezione [default 1]: 1 ②
Utente: MarkPilgrim ③
Password:
Registro chardet su http://pypi.python.org/pypi ④
Risposta del server (200): OK
eseguo il comando sdist ⑤
... messaggi omessi per brevità ...
eseguo il comando bdist_wininst ⑥
... messaggi omessi per brevità ...
eseguo il comando upload ⑦
Invio dist\chardet-1.0.2.zip a http://pypi.python.org/pypi
Risposta del server (200): OK
Invio dist\chardet-1.0.2.win32.exe a http://pypi.python.org/pypi
Risposta del server (200): OK
Posso memorizzare i tuoi dati di accesso su PyPI per rendere più veloci le future trasmissioni.
(i dati verranno memorizzati in c:\home\pypirc)
Salvare i tuoi dati di accesso (y/N)? n ⑧

```

1. Quando rilasciate il vostro progetto per la prima volta, Distutils aggiungerà il vostro software all'Indice dei Pacchetti Python e gli darà un proprio URL. Tutte le volte successive si limiterà ad aggiornare i metadati del progetto con qualsiasi cambiamento possiate aver apportato ai parametri del vostro file setup.py. Poi, assemblerà una distribuzione dei sorgenti (sdist) e un pacchetto di installazione per Windows (bdist_wininst) e li caricherà entrambi su PyPI (upload).
2. Digitate 1 o premete semplicemente INVIO per selezionare “usa i tuoi dati di accesso esistenti”.

3. Inserite il nome utente e la password che avete selezionato sulla pagina di registrazione per gli utenti di PyPI. Distutils non mostrerà la vostra password sullo schermo, né mostrerà asterischi al posto dei caratteri. Digitate semplicemente la vostra password e premete INVIO.
4. Distutils registra il vostro pacchetto sull'Indice dei Pacchetti Python...
5. ...assembla la vostra distribuzione dei sorgenti...
6. ...crea il pacchetto di installazione per Windows...
7. ...e li carica entrambi sull'Indice dei Pacchetti Python.
8. Se volete automatizzare il processo di rilascio delle nuove versioni, avrete bisogno di salvare le vostre credenziali su PyPI in un file locale. Questo è assolutamente insicuro e completamente opzionale.

Congratulazioni, ora avete la vostra pagina sull'Indice dei Pacchetti Python! L'indirizzo è

<http://pypi.python.org/pypi/NOME>, dove *NOME* è la stringa che avete passato nel parametro name nel vostro file setup.py.

Se volete rilasciare una nuova versione, vi basta aggiornare il vostro file setup.py con il nuovo numero di versione e invocare lo stesso comando di prima:

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py register sdist bdist_wininst upload
```



16.11. I MOLTI POSSIBILI FUTURI DELLA DISTRIBUZIONE DI LIBRERIE PYTHON

Distutils non è la quintessenza della distribuzione di librerie Python, ma al momento della scrittura (agosto 2009) è l'unico framework per la creazione di pacchetti che funziona con Python 3. Ci sono un certo numero di altri framework per Python 2: alcuni si concentrano sull'installazione, altri sul collaudo e il deployment. In futuro, alcuni o persino tutti potrebbero venire convertiti verso Python 3.

Questi framework si concentrano sull'installazione:

- Setuptools

- [Pip](#)
- [Distribute](#)

Questi si concentrano sul collaudo e il deployment:

- [virtualenv](#)
- [zc.buildout](#)
- [Paver](#)
- [Fabric](#)
- [py2exe](#)



16.12. LETTURE DI APPROFONDIMENTO

Su Distutils:

- [Distribuire moduli Python con Distutils](#)
- [Le funzionalità principali di Distutils](#) elenca tutti i possibili argomenti per la funzione `setup()`
- [Il ricettario di Distutils](#)
- [PEP 370: directory site-packages per il singolo utente](#)
- [La PEP 370 e lo “stufato d’ambiente”](#)

Su altri framework per la creazione di pacchetti:

- [L’ecosistema della creazione di pacchetti Python](#)
- [Sulla distribuzione](#)
- [Alcune correzioni a “Sulla distribuzione”](#)
- [Perché mi piace Pip](#)
- [La distribuzione di pacchetti Python: alcune osservazioni](#)
- [Nessuno si aspetta la distribuzione di pacchetti Python!](#)

APPENDICE A. CONVERTIRE CODICE VERSO PYTHON 3 CON 2to3

“ La vita è piacevole. La morte è pacifica. È la transizione che crea dei problemi. ”

— Isaac Asimov (attribuita)

A.1. IMMERSIONE!

Una quantità talmente grande di caratteristiche del linguaggio è cambiata tra Python 2 e Python 3 che solo un numero molto ridotto e sempre più esiguo di programmi verrà eseguito senza modifiche da entrambi gli interpreti. Per facilitare questa transizione, Python 3 include uno script di utilità chiamato 2to3, che prende in ingresso il vostro codice sorgente in Python 2 e lo converte automaticamente verso Python 3 tanto quanto gli è possibile. Il [Caso di studio: convertire chardet verso Python 3](#) descrive come eseguire 2to3, poi mostra alcune cose che lo script non riesce a correggere automaticamente. Questa appendice illustra ciò che lo script riesce a correggere automaticamente

A.2. L'ISTRUZIONE print

In Python 2 `print` era un'istruzione. Per stampare una cosa qualsiasi bastava farla seguire alla parola chiave `print`. In Python 3 [`print\(\)`](#) è una funzione. Qualunque cosa vogliate stampare, passatela a `print()` nel modo in cui la passate a qualsiasi altra funzione.

Note	Python 2	Python 3
①	<code>print</code>	<code>print()</code>
②	<code>print 1</code>	<code>print(1)</code>
③	<code>print 1, 2</code>	<code>print(1, 2)</code>
④	<code>print 1, 2,</code>	<code>print(1, 2, end=' ')</code>
⑤	<code>print >>sys.stderr, 1, 2, 3</code>	<code>print(1, 2, 3, file=sys.stderr)</code>

1. Per stampare una riga vuota, invocate `print()` senza alcun argomento.
2. Per stampare un singolo valore, invocate `print()` con un argomento.
3. Per stampare due valori separati da uno spazio, invocate `print()` con due argomenti.
4. Questo può risultare ingannevole. In Python 2 un'istruzione `print` conclusa da una virgola avrebbe stampato i valori separati da spazi, seguiti da uno spazio finale, evitando di stampare un ritorno a capo. (Tecnicamente, il funzionamento di `print` è più complicato. In Python 2 l'istruzione `print` usava un attributo chiamato `softspace`, ora deprecato. Invece di stampare uno spazio, Python 2 impostava `sys.stdout.softspace` a 1, rimandando la stampa di quel carattere fino a quando non veniva stampato qualcos'altro sulla stessa riga. Se la successiva istruzione `print` stampava un ritorno a capo, `sys.stdout.softspace` veniva impostato a 0 e lo spazio non sarebbe comparso. Probabilmente non avreste mai notato la differenza a meno che la vostra applicazione non fosse stata sensibile alla presenza o all'assenza di spazi bianchi alla fine di una riga nel testo generato da `print`.) Per ottenere questo comportamento in Python 3 è necessario passare `end=' '` come argomento con nome alla funzione `print()`. L'argomento `end` assume `'\n'` (un ritorno a capo) come valore predefinito, quindi va modificato per sopprimere il ritorno a capo dopo la stampa degli altri argomenti.
5. In Python 2 potevate redirigere l'uscita verso un canale — come `sys.stderr` — utilizzando la sintassi `>>nome_del_canale`. Per ottenere la redirectione in Python 3 è necessario passare il canale come valore dell'argomento con nome `file`. L'argomento `file` assume `sys.stdout` (uscita standard) come valore predefinito, quindi va modificato per produrre l'uscita su un canale differente.

A.3. LETTERALI STRINGA UNICODE

Python 2 aveva due tipi di stringa: stringhe Unicode e stringhe non Unicode. Python 3 ha un solo tipo di stringa: stringhe Unicode.

Note	Python 2	Python 3
①	<code>u'PapayaWhip'</code>	<code>'PapayaWhip'</code>
②	<code>ur'PapayaWhip\foo'</code>	<code>r'PapayaWhip\foo'</code>

1. I letterali stringa Unicode sono semplicemente convertiti in letterali stringa, che in Python 3 sono sempre di tipo Unicode.
2. Le *stringhe raw* Unicode (per le quali Python non effettua l'escape automatico dei backslash) vengono convertite in stringhe raw. In Python 3, le stringhe raw sono sempre di tipo Unicode.

A.4. LA FUNZIONE GLOBALE `unicode()`

Python 2 aveva due funzioni globali per convertire oggetti in stringhe: `unicode()` per convertirli in stringhe Unicode e `str()` per convertirli in stringhe non Unicode. Python 3 ha un solo tipo di stringhe, stringhe Unicode, quindi la funzione `str()` è tutto quello che vi serve. (La funzione `unicode()` non esiste più.)

Note	Python 2	Python 3
	<code>unicode(anything)</code>	<code>str(anything)</code>

A.5. IL TIPO DI DATO `long`

Python 2 aveva tipi `int` e `long` separati per i numeri non in virgola mobile. Un `int` non poteva essere più grande di `sys.maxint`, il cui valore variava a seconda della piattaforma. I `long` venivano definiti aggiungendo una `L` alla fine del numero e potevano essere, be', più lunghi degli `int`. In Python 3 c'è solo un tipo di numeri interi chiamato `int`, che si comporta quasi sempre come il tipo `long` in Python 2. Dato che non ci sono più due tipi, non c'è alcun bisogno di una sintassi speciale per distinguerli.

Per approfondire l'argomento, si consiglia la lettura della PEP 237: Unificare gli interi lunghi e gli interi.

Note	Python 2	Python 3
①	<code>x = 1000000000000L</code>	<code>x = 1000000000000</code>
②	<code>x = 0xFFFFFFFFFFFL</code>	<code>x = 0xFFFFFFFFFFFF</code>
③	<code>long(x)</code>	<code>int(x)</code>
④	<code>type(x) is long</code>	<code>type(x) is int</code>
⑤	<code>isinstance(x, long)</code>	<code>isinstance(x, int)</code>

1. I letterali `long` in base 10 diventano letterali `int` in base 10.
2. I letterali `long` in base 16 diventano letterali `int` in base 16.
3. In Python 3 la vecchia funzione `long()` non esiste più, dato che i `long` non esistono più. Per convertire una variabile in un intero, usate la funzione `int()`.
4. Per controllare se una variabile è un intero, recuperatene il tipo e confrontatelo con `int`, non con `long`.
5. Potete anche usare la funzione `isinstance()` per controllare i tipi di dato; usate ancora `int`, non `long`, per controllare gli interi.

A.6. IL CONFRONTO DI DISUGUAGLIANZA CON <>

Python 2 supportava <> come sinonimo di !=, l'operatore per il confronto di disuguaglianza. Python 3 supporta l'operatore !=, ma non <>.

Note	Python 2	Python 3
①	<code>if x <> y:</code>	<code>if x != y:</code>
②	<code>if x <> y <> z:</code>	<code>if x != y != z:</code>

1. Un semplice confronto.
2. Un confronto più complesso tra tre valori.

A.7. IL METODO `has_key()` PER I DIZIONARI

In Python 2 i dizionari avevano un metodo `has_key()` per verificare che il dizionario contenesse una certa chiave. In Python 3 questo metodo non esiste più. Al suo posto dovete usare l'operatore `in`.

Note	Python 2	Python 3
①	<code>a_dictionary.has_key('PapayaWhip')</code>	<code>'PapayaWhip' in a_dictionary</code>
②	<code>a_dictionary.has_key(x) or a_dictionary.has_key(y)</code>	<code>x in a_dictionary or y in a_dictionary</code>
③	<code>a_dictionary.has_key(x or y)</code>	<code>(x or y) in a_dictionary</code>
④	<code>a_dictionary.has_key(x + y)</code>	<code>(x + y) in a_dictionary</code>
⑤	<code>x + a_dictionary.has_key(y)</code>	<code>x + (y in a_dictionary)</code>

1. La forma più semplice.
2. L'operatore `in` ha la precedenza sull'operatore `or`, quindi in questo caso non c'è alcun bisogno di usare le parentesi attorno a `x in a_dictionary` o a `y in a_dictionary`.
3. D'altra parte, in questo caso avete bisogno di usare le parentesi attorno a `x or y`, per lo stesso motivo — `in` ha la precedenza su `or`. Notate che questo codice è completamente differente da quello nella riga precedente. Python interpreta prima `x or y`, che dà come risultato `x` (se `x` è vero in un contesto logico) o `y`; poi prende quel singolo valore e controlla se è una chiave di `a_dictionary`.
4. L'operatore `+` ha la precedenza sull'operatore `in`, quindi questa forma non avrebbe tecnicamente bisogno delle parentesi attorno a `x + y`, ma 2to3 le aggiunge comunque.

5. Questa forma ha certamente bisogno delle parentesi attorno a `y` in `a_dictionary`, dato che l'operatore `+` ha la precedenza sull'operatore `in`.

A.8. METODI DEI DIZIONARI CHE RESTITUISCONO LISTE

In Python 2 molti metodi dei dizionari restituivano liste. I metodi più frequentemente usati erano `keys()`, `items()` e `values()`. In Python 3 tutti questi metodi restituiscono viste dinamiche. In alcuni contesti questo non è un problema. Se il valore di ritorno del metodo viene immediatamente passato a un'altra funzione che itera attraverso l'intera sequenza, non fa alcuna differenza che il tipo reale di quel valore sia una lista o una vista. In altri contesti ha invece molta importanza. Se vi stavate aspettando una lista completa con elementi accessibili individualmente, il vostro programma si bloccherà perché le viste non supportano l'accesso basato su indici.

Note	Python 2	Python 3
①	<code>a_dictionary.keys()</code>	<code>list(a_dictionary.keys())</code>
②	<code>a_dictionary.items()</code>	<code>list(a_dictionary.items())</code>
③	<code>a_dictionary.iterkeys()</code>	<code>iter(a_dictionary.keys())</code>
④	<code>[i for i in a_dictionary.iterkeys()]</code>	<code>[i for i in a_dictionary.keys()]</code>
⑤	<code>min(a_dictionary.keys())</code>	<i>nessuna modifica</i>

1. 2to3 pecca per eccesso di prudenza, convertendo il valore di ritorno di `keys()` in una lista statica tramite la funzione `list()`. Questo codice funzionerà sempre, ma sarà meno efficiente rispetto all'uso di una vista. Dovreste esaminare il codice convertito per controllare se una lista è assolutamente necessaria o se una vista sarebbe sufficiente.
2. Un'altra conversione di vista in lista, questa volta con il metodo `items()`. 2to3 farà la stessa cosa con il metodo `values()`.
3. Python 3 non supporta più il metodo `iterkeys()`. Usate `keys()` e, se necessario, convertite la vista in un iteratore tramite la funzione `iter()`.
4. 2to3 riconosce quando il metodo `iterkeys()` viene usato in una descrizione di lista e lo converte nel metodo `keys()` (senza circondarlo con una chiamata aggiuntiva a `iter()`). Questa modifica funziona perché le viste sono iterabili.
5. 2to3 riconosce che il risultato restituito dal metodo `keys()` viene immediatamente passato a una funzione che itera su un'intera sequenza, così non ha bisogno di convertirne il valore in una lista. La funzione `min()`

itererà tranquillamente attraverso la vista. Queste considerazioni valgono per le funzioni `min()`, `max()`, `sum()`, `list()`, `tuple()`, `set()`, `sorted()`, `any()` e `all()`.

A.9. MODULI CHE SONO STATI RINOMINATI O RIORGANIZZATI

Diversi moduli nella libreria standard di Python sono stati rinominati. Diversi altri moduli in relazione tra loro sono stati combinati o riorganizzati per rendere le loro associazioni più logiche.

A.9.1. http

In Python 3 molti moduli relativi al protocollo HTTP sono stati combinati in un unico pacchetto chiamato `http`.

Note	Python 2	Python 3
①	<code>import urllib</code>	<code>import http.client</code>
②	<code>import Cookie</code>	<code>import http.cookies</code>
③	<code>import cookielib</code>	<code>import http.cookiejar</code>
④	<code>import BaseHTTPServer</code> <code>import SimpleHTTPServer</code> <code>import CGIHTTPServer</code>	<code>import http.server</code>

1. Il modulo `http.client` implementa una libreria di basso livello che può richiedere risorse HTTP e interpretare le risposte HTTP.
2. Il modulo `http.cookies` fornisce un'interfaccia Python ai cookie inviati a un browser tramite l'intestazione `HTTP Set-Cookie:`.
3. Il modulo `http.cookiejar` manipola gli effettivi file usati dai browser web più popolari per memorizzare i cookie.
4. Il modulo `http.server` fornisce un server HTTP di base.

A.9.2. urllib

Python 2 aveva un macello di moduli sovrapposti per riconoscere, codificare e recuperare gli URL. In Python 3 tutti questi moduli sono stati riorganizzati e combinati in un singolo pacchetto chiamato `urllib`.

Note	Python 2	Python 3
①	<code>import urllib</code>	<code>import urllib.request, urllib.parse, urllib.error</code>
②	<code>import urllib2</code>	<code>import urllib.request, urllib.error</code>
③	<code>import urlparse</code>	<code>import urllib.parse</code>
④	<code>import robotparser</code>	<code>import urllib.robotparser</code>
⑤	<code>from urllib import FancyURLopener</code> <code>from urllib import urlencode</code>	<code>from urllib.request import FancyURLopener</code> <code>from urllib.parse import urlencode</code>
⑥	<code>from urllib2 import Request</code> <code>from urllib2 import HTTPError</code>	<code>from urllib.request import Request</code> <code>from urllib.error import HTTPError</code>

1. Il vecchio modulo `urllib` di Python 2 raccoglieva una varietà di funzioni, comprese `urlopen()` per recuperare dati e `splitttype()`, `splithost()` e `splituser()` per suddividere un URL nelle sue parti costituenti. Queste funzioni sono state riorganizzate più logicamente nell'ambito del nuovo pacchetto `urllib`. 2to3 modificherà anche tutte le chiamate a queste funzioni in modo da usare il nuovo schema di nomi.
2. Il vecchio modulo `urllib2` di Python 2 è stato incorporato nel pacchetto `urllib` di Python 3. Tutte le vostre funzionalità di `urllib2` preferite — il metodo `build_opener()`, gli oggetti `Request`, `HTTPBasicAuthHandler` e compagnia — sono ancora disponibili.
3. Il modulo `urllib.parse` di Python 3 contiene tutte le funzioni di riconoscimento del vecchio modulo `urlparse` di Python 2.
4. Il modulo `urllib.robotparser` riconosce i file `robots.txt`.
5. La classe `FancyURLopener` che gestisce le redirezioni HTTP e altri codici di stato è ancora disponibile nel nuovo modulo `urllib.request`. La funzione `urlencode()` è stata spostata in `urllib.parse`.
6. L'oggetto `Request` è ancora disponibile in `urllib.request`, ma costanti come `HTTPError` sono state spostate in `urllib.error`.

Vi ho detto che 2to3 riscriverà anche le vostre chiamate di funzione? Per esempio, se il vostro codice in Python 2 importa il modulo `urllib` e invoca `urllib.urlopen()` per prelevare dati, 2to3 correggerà sia l'istruzione di importazione che la chiamata di funzione.

Note	Python 2	Python 3
	<code>import urllib</code> <code>print urllib.urlopen('http://diveintopython3.org/').read()</code>	<code>import urllib.request, urllib.parse,</code> <code>print(urllib.request.urlopen('http://</code>

A.9.3. dbm

Tutti i vari cloni di DBM si trovano ora in un singolo pacchetto chiamato `dbm`. Se avete bisogno di una specifica variante come GNU DBM potete importare il modulo appropriato nell'ambito del pacchetto `dbm`.

Note	Python 2	Python 3
	<code>import dbm</code>	<code>import dbm.ndbm</code>
	<code>import gdbm</code>	<code>import dbm.gnu</code>
	<code>import dbhash</code>	<code>import dbm.bsd</code>
	<code>import dumbdbm</code>	<code>import dbm.dumb</code>
	<code>import anydbm</code> <code>import whichdb</code>	<code>import dbm</code>

A.9.4. xmlrpc

XML-RPC è un metodo leggero per eseguire chiamate RPC remote su HTTP. La libreria client di XML-RPC e diverse implementazioni server di XML-RPC sono ora combinate in un singolo pacchetto chiamato `xmlrpc`.

Note	Python 2	Python 3
	<code>import xmlrpclib</code>	<code>import xmlrpc.client</code>
	<code>import DocXMLRPCServer</code> <code>import SimpleXMLRPCServer</code>	<code>import xmlrpc.server</code>

A.9.5. ALTRI MODULI

Note	Python 2	Python 3
①	<code>try:</code> <code>import cStringIO as StringIO</code> <code>except ImportError:</code> <code>import StringIO</code>	<code>import io</code>
②	<code>try:</code> <code>import cPickle as pickle</code> <code>except ImportError:</code> <code>import pickle</code>	<code>import pickle</code>

③	<code>import __builtin__</code>	<code>import builtins</code>
④	<code>import copy_reg</code>	<code>import copyreg</code>
⑤	<code>import Queue</code>	<code>import queue</code>
⑥	<code>import SocketServer</code>	<code>import socketserver</code>
⑦	<code>import ConfigParser</code>	<code>import configparser</code>
⑧	<code>import repr</code>	<code>import reprlib</code>
⑨	<code>import commands</code>	<code>import subprocess</code>

1. Un idioma comune in Python 2 era quello di importare `cStringIO` as `StringIO` e, in caso di errore, importare `StringIO` al suo posto. Non è necessario farlo in Python 3, poiché il modulo `io` lo farà per voi: troverà l'implementazione più veloce disponibile e la utilizzerà automaticamente.
2. Un idioma simile veniva usato per importare l'implementazione più veloce di `pickle`, il protocollo di serializzazione degli oggetti. Non è necessario usare tale idioma in Python 3, poiché il modulo `pickle` lo farà per voi.
3. Il modulo `builtins` contiene le funzioni globali, le classi e le costanti usate in tutto il linguaggio Python. Ridefinire una funzione nel modulo `builtins` ridefinirà la funzione globale ovunque. Questo è esattamente tanto potente e spaventoso quanto sembra.
4. Il modulo `copyreg` aggiunge il supporto per la serializzazione ai vostri tipi di dato definiti in C.
5. Il modulo `queue` implementa una coda che supporta molteplici produttori e consumatori.
6. Il modulo `socketserver` fornisce classi base generiche per implementare differenti tipi di server per le socket.
7. Il modulo `configparser` riconosce i file di configurazione sullo stile dei file INI.
8. Il modulo `reprlib` reimplementa la funzione built-in `repr()` introducendo controlli aggiuntivi su quanto possono essere lunghe le rappresentazioni prima di venire troncate.
9. Il modulo `subprocess` vi permette di creare processi, connettervi ai loro canali e ottenere i loro codici di ritorno.

A.10. IMPORTAZIONI RELATIVE ALL'INTERNO DI UN PACCHETTO

Un pacchetto è un gruppo di moduli correlati che funziona come una singola entità. In Python 2, quando i moduli all'interno di un pacchetto devono fare riferimento gli uni agli altri, si usa l'idioma `import foo` oppure `from foo import Bar`. L'interprete Python 2 prima cerca all'interno del pacchetto corrente per trovare `foo.py`, poi si sposta nelle altre directory contenute nel percorso di ricerca di Python (`sys.path`). Python 3 funziona in modo leggermente diverso. Invece di cercare nel pacchetto corrente, si sposta direttamente sul

percorso di ricerca di Python. Se volete che un modulo in un pacchetto importi un altro modulo nello stesso pacchetto, dovete esplicitamente fornire il percorso relativo tra i due moduli.

Supponete di avere questo pacchetto, contenente più file nella stessa directory:

```
chardet/
|
+--__init__.py
|
+--constants.py
|
+--mbcharsetprober.py
|
+--universaldetector.py
```

Ora supponete che `universaldetector.py` abbia bisogno di importare l'intero file `constants.py` e una classe da `mbcharsetprober.py`. Come fareste?

Note	Python 2	Python 3
①	<code>import constants</code>	<code>from . import constants</code>
②	<code>from mbcharsetprober import MultiByteCharSetProber</code>	<code>from .mbcharsetprober import MultiByteCharsetProber</code>

- 1. Quando dovete importare un intero modulo in qualche punto del vostro pacchetto, usate la nuova sintassi `from . import`. Il punto è in realtà il percorso relativo da questo file (`universaldetector.py`) al file che volete importare (`constants.py`). In questo caso sono nella stessa directory, quindi è sufficiente un singolo punto. Potete anche importare dalla directory superiore (`from .. import altromodulo`) o da una sottodirectory.
- 2. Per importare una specifica classe o funzione da un altro modulo direttamente nello spazio di nomi del vostro modulo, fate precedere al nome del modulo obiettivo un prefisso contenente un percorso relativo meno lo slash finale. In questo caso `mbcharsetprober.py` è nella stessa directory di `universaldetector.py`, quindi il percorso è un singolo punto. Potete anche importare dalla directory genitore (`from ..altromodulo import AltraClasse`) o da una sottodirectory.

A.11. IL METODO `next()` DEGLI ITERATORI

In Python 2 gli iteratori hanno un metodo `next()` che restituisce l'elemento successivo nella sequenza. Questo è ancora vero in Python 3, ma esiste anche una funzione globale `next()` che prende un iteratore come argomento.

Note	Python 2	Python 3
①	<code>anIterator.next()</code>	<code>next(anIterator)</code>
②	<code>a_function_that_returns_an_iterator().next()</code>	<code>next(a_function_that_returns_an_iterator())</code>
③	<pre>class A: def next(self): pass</pre>	<pre>class A: def __next__(self): pass</pre>
④	<pre>class A: def next(self, x, y): pass</pre>	<i>nessuna modifica</i>
⑤	<pre>next = 42 for an_iterator in a_sequence_of_iterators: an_iterator.next()</pre>	<pre>next = 42 for an_iterator in a_sequence_of_iterators: an_iterator.__next__()</pre>

1. Nel caso più semplice, invece di chiamare il metodo `next()` di un iteratore ora passate l'iteratore stesso alla funzione globale `next()`.
2. Se avete una funzione che restituisce un iteratore, chiamate la funzione e passate il risultato alla funzione `next()`. (Lo script 2to3 è abbastanza intelligente da convertire questi casi in maniera appropriata.)
3. Se definite una vostra classe e volete usarla come un iteratore, definite il metodo speciale `__next__()`.
4. Se definite una vostra classe e la dotate di un metodo chiamato `next()` che prende uno o più argomenti, 2to3 non lo toccherà. Questa classe non può essere usata come un iteratore, perché il suo metodo `next()` accetta un certo numero di argomenti.
5. Questo caso è un po' complicato. Se avete una variabile locale chiamata `next`, allora essa ha precedenza sulla funzione globale `next()`. Perciò, siete obbligati a chiamare il metodo speciale `__next__()` dell'iteratore per ottenere l'elemento successivo nella sequenza. (In alternativa potreste anche modificare il codice in modo che la variabile locale non sia più chiamata `next`, ma 2to3 non lo farà automaticamente per voi.)

A.12. LA FUNZIONE GLOBALE filter()

In Python 2 la funzione `filter()` restituiva una lista come risultato del filtraggio di una sequenza attraverso una funzione che restituisce `True` o `False` per ogni elemento della sequenza. In Python 3 la funzione `filter()` restituisce un iteratore, non una lista.

Note	Python 2	Python 3
①	<code>filter(a_function, a_sequence)</code>	<code>list(filter(a_function, a_sequence))</code>
②	<code>list(filter(a_function, a_sequence))</code>	<i>nessuna modifica</i>
③	<code>filter(None, a_sequence)</code>	<code>[i for i in a_sequence if i]</code>
④	<code>for i in filter(None, a_sequence):</code>	<i>nessuna modifica</i>
⑤	<code>[i for i in filter(a_function, a_sequence)]</code>	<i>nessuna modifica</i>

1. Nel caso più basilare, 2to3 circonda la chiamata a `filter()` con una chiamata a `list()`, che semplicemente itera attraverso il suo argomento e restituisce una vera lista.
2. Comunque, se la chiamata a `filter()` è già circondata da `list()` allora 2to3 non farà nulla, in quanto è irrilevante che `filter()` restituisca un iteratore.
3. Per la sintassi speciale di `filter(None, ...)`, 2to3 trasformerà la chiamata in una descrizione di lista semanticamente equivalente.
4. In contesti come i cicli `for`, che iterano comunque attraverso l'intera sequenza, non sono necessari cambiamenti.
5. Anche in questo caso non sono necessari cambiamenti, perché la descrizione di lista itererà attraverso l'intera sequenza, potendolo fare altrettanto bene sia che `filter()` restituisca un iteratore sia che restituisca una lista.

A.13. LA FUNZIONE GLOBALE map()

In maniera molto simile a `filter()`, la funzione `map()` ora restituisce un iteratore. (In Python 2 restituiva una lista.)

Note	Python 2	Python 3
①	<code>map(a_function, 'PapayaWhip')</code>	<code>list(map(a_function, 'PapayaWhip'))</code>
②	<code>map(None, 'PapayaWhip')</code>	<code>list('PapayaWhip')</code>

③	<code>map(lambda x: x+1, range(42))</code>	<code>[x+1 for x in range(42)]</code>
④	<code>for i in map(a_function, a_sequence):</code>	<i>nessuna modifica</i>
⑤	<code>[i for i in map(a_function, a_sequence)]</code>	<i>nessuna modifica</i>

1. Come con `filter()`, nel caso più basilare 2to3 circonda la chiamata a `map()` con una chiamata a `list()`.
2. La sintassi speciale di `map(None, ...)`, cioè la funzione identità, sarà convertita da 2to3 in una chiamata a `list()` equivalente.
3. Se il primo argomento di `map()` è una funzione lambda, 2to3 la convertirà in una descrizione di lista semanticamente equivalente.
4. In contesti come i cicli `for`, che iterano comunque attraverso l'intera sequenza, non sono necessari cambiamenti.
5. Anche in questo caso non sono necessari cambiamenti, perché la descrizione di lista itererà attraverso l'intera sequenza, potendolo fare altrettanto bene sia che `map()` restituisca un iteratore sia che restituisca una lista.

A.14. LA FUNZIONE GLOBALE `reduce()`

In Python 3 la funzione `reduce()` è stata rimossa dallo spazio di nomi globale e collocata nel modulo `functools`.

Note	Python 2	Python 3
	<code>reduce(a, b, c)</code>	<code>from functools import reduce</code> <code>reduce(a, b, c)</code>

A.15. LA FUNZIONE GLOBALE `apply()`

Python 2 aveva una funzione globale chiamata `apply()`, che prende una funzione `f` e una lista `[a, b, c]` e restituisce `f(a, b, c)`. Potete ottenere la stessa cosa chiamando la funzione direttamente e passandole la lista degli argomenti preceduta da un asterisco. In Python 3 la funzione `apply()` non esiste più, perciò dovete usare la notazione con l'asterisco.

Note	Python 2	Python 3
①	<code>apply(a_function, a_list_of_args)</code>	<code>a_function(*a_list_of_args)</code>

②	<code>apply(a_function, a_list_of_args, a_dictionary_of_named_args)</code>	<code>a_function(*a_list_of_args, **a_dictionary_of_named_args)</code>
③	<code>apply(a_function, a_list_of_args + z)</code>	<code>a_function(*a_list_of_args + z)</code>
④	<code>apply(aModule.a_function, a_list_of_args)</code>	<code>aModule.a_function(*a_list_of_args)</code>

1. Nella forma più semplice, potete chiamare una funzione con una lista di argomenti (una lista reale come [a, b, c]) facendo precedere la lista da un asterisco (*). Questa forma è esattamente equivalente alla vecchia funzione `apply()` di Python 2.
2. In Python 2 la funzione `apply()` poteva in realtà accettare tre parametri: una funzione, una lista di argomenti e un dizionario di argomenti con nome. In Python 3 potete ottenere la stessa cosa facendo precedere la lista di argomenti da un asterisco (*) e il dizionario di argomenti con nome da due asterischi (**).
3. L'operatore +, usato qui per concatenare liste, ha precedenza sull'operatore *, quindi non c'è bisogno di parentesi aggiuntive attorno ad `a_list_of_args + z`.
4. Lo script `2to3` è abbastanza scaltro da convertire chiamate complesse ad `apply()`, comprese chiamate a funzioni contenute in moduli importati.

A.16. LA FUNZIONE GLOBALE `intern()`

In Python 2 potevate chiamare la funzione `intern()` su una stringa per memorizzarla in una tabella interna all'interprete allo scopo di ottimizzare le prestazioni. In Python 3 la funzione `intern()` è stata spostata nel modulo `sys`.

Note	Python 2	Python 3
	<code>intern(aString)</code>	<code>sys.intern(aString)</code>

A.17. L'ISTRUZIONE `exec`

Esattamente come l'istruzione `print` è diventata una funzione in Python 3, così è successo anche all'istruzione `exec`. La funzione `exec()` prende una stringa che contiene codice Python arbitrario e la esegue come se fosse un'altra istruzione o espressione. `exec()` è come `eval()`, ma ancora più potente e pericolosa. La funzione `eval()` può solo valutare una singola espressione, ma `exec()` può eseguire molteplici istruzioni, importazioni, dichiarazioni di funzione — essenzialmente un intero programma Python in una stringa.

Note	Python 2	Python 3
------	----------	----------

①	<code>exec codeString</code>	<code>exec(codeString)</code>
②	<code>exec codeString in a_global_namespace</code>	<code>exec(codeString, a_global_namespace)</code>
③	<code>exec codeString in a_global_namespace, a_local_namespace</code>	<code>exec(codeString, a_global_namespace, a_local_namespace)</code>

1. Nella sua forma più semplice, lo script 2to3 racchiude semplicemente tra parentesi il codice sotto forma di stringa, dato che `exec()` è ora una funzione invece di un'istruzione.
2. La vecchia istruzione `exec` poteva utilizzare uno spazio di nomi per rappresentare un ambiente privato di nomi globali nel quale il codice sotto forma di stringa sarebbe stato eseguito. Per fare la stessa cosa in Python 3 basta passare lo spazio di nomi alla funzione `exec()` come secondo argomento.
3. In maniera ancora più elaborata, la vecchia istruzione `exec` poteva anche utilizzare uno spazio di nomi locali (come le variabili definite nell'ambito di una funzione). In Python 3 la funzione `exec()` può fare anche questo.

A.18. L'ISTRUZIONE `execfile`

Come la vecchia istruzione `exec`, anche la vecchia istruzione `execfile` esegue stringhe come se fossero codice Python. Laddove `exec` prendeva una stringa, `execfile` prendeva un nome di file. In Python 3 l'istruzione `execfile` è stata eliminata. Se avete davvero bisogno di eseguire un file di codice Python (ma non siete disposti semplicemente a importarlo) potete ottenere lo stesso effetto aprendo il file, leggendone il contenuto, chiamando la funzione globale `compile()` per obbligare l'interprete Python a compilare il codice e infine utilizzando la nuova funzione `exec()`.

Note	Python 2	Python 3
	<code>execfile('nome_di_file')</code>	<code>exec(compile(open('nome_di_file').read(), 'nome_di_file', 'exec'))</code>

A.19. I LETTERALI `repr` (I BACKTICK)

Python 2 utilizzava una speciale sintassi per ottenere una rappresentazione di qualsiasi oggetto racchiudendo quell'oggetto tra backtick (come ``x``). In Python 3 questa possibilità esiste ancora, ma dovete usare la funzione globale `repr()` al posto dei backtick.

Note	Python 2	Python 3
①	<code>`x`</code>	<code>repr(x)</code>

②	<code>`'PapayaWhip' + `2``</code>	<code>repr('PapayaWhip' + repr(2))</code>
---	-----------------------------------	---


1. Ricordate, `x` può essere qualsiasi cosa — una classe, una funzione, un modulo, un tipo di dato primitivo, &c. La funzione `repr()` lavora su tutto.
2. In Python 2 i backtick potevano essere annidati, producendo questo tipo di espressioni confuse (ma valide). Lo strumento `2to3` è abbastanza scaltro da convertire questi casi in chiamate annidate a `repr()`.

A.20. L'ISTRUZIONE `try...except`

La sintassi per catturare le eccezioni è leggermente cambiata tra Python 2 e Python 3.

Note	Python 2	Python 3
①	<pre>try: import mymodule except ImportError, e: pass</pre>	<pre>try: import mymodule except ImportError as e: pass</pre>
②	<pre>try: import mymodule except (RuntimeError, ImportError), e: pass</pre>	<pre>try: import mymodule except (RuntimeError, ImportError) as e: pass</pre>
③	<pre>try: import mymodule except ImportError: pass</pre>	<i>no change</i>
④	<pre>try: import mymodule except: pass</pre>	<i>no change</i>

1. Invece di una virgola dopo il tipo dell'eccezione, Python 3 usa la nuova parola chiave `as`.
2. La parola chiave `as` funziona anche per catturare più tipi di eccezione alla volta.
3. Se catturate un'eccezione ma in realtà non vi interessa accedere all'oggetto eccezione in sé, la sintassi è identica tra Python 2 e Python 3.
4. Similmente, se usate un'istruzione `except` vuota per catturare *tutte* le eccezioni, la sintassi è identica.

 Non dovrete mai usare un'istruzione `except` vuota per catturare *tutte* le eccezioni quando importate moduli (o nella maggior parte delle altre occasioni). Se lo fate, rischiate di catturare anche eccezioni come `KeyboardInterrupt` (se l'utente ha premuto `Ctrl-C` per interrompere il programma) rendendo in questo modo molto più difficile effettuare il debug degli errori.

A.21. L'ISTRUZIONE `raise`

La sintassi per sollevare le vostre eccezioni è leggermente cambiata tra Python 2 e Python 3.

Note	Python 2	Python 3
①	<code>raise MyException</code>	<i>nessuna modifica</i>
②	<code>raise MyException, 'messaggio di errore'</code>	<code>raise MyException('messaggio di errore')</code>
③	<code>raise MyException, 'messaggio di errore', a_traceback</code>	<code>raise MyException('messaggio di errore').with_traceback(a_traceback)</code>
④	<code>raise 'messaggio di errore'</code>	<i>non supportato</i>

1. Nella sua forma più semplice, cioè sollevare un'eccezione senza un messaggio personalizzato, la sintassi è rimasta invariata.
2. Il cambiamento si nota quando volete sollevare un'eccezione con un messaggio personalizzato. Python 2 separava con una virgola la classe dell'eccezione e il messaggio, mentre Python 3 passa il messaggio di errore come un parametro.
3. Python 2 supportava una sintassi più complessa per sollevare un'eccezione con una *traceback* (la traccia dello stack di esecuzione) personalizzata. Potete farlo anche in Python 3, ma la sintassi è abbastanza differente.
4. In Python 2 potevate sollevare un'eccezione senza alcuna classe, utilizzando semplicemente un messaggio di errore. In Python 3 questo non è più possibile. `2to3` vi avvertirà che non è in grado di correggere automaticamente questo caso.

A.22. IL METODO `throw` DEI GENERATORI

In Python 2 i generatori avevano un metodo `throw()`. Chiamare `a_generator.throw()` solleva un'eccezione nel punto in cui il generatore è stato sospeso, poi restituisce il valore successivo prodotto dalla funzione generatrice. In Python 3 questa funzionalità è ancora disponibile, ma la sintassi è leggermente diversa.

Note	Python 2	Python 3
①	<code>a_generator.throw(MyException)</code>	<i>nessuna modifica</i>
②	<code>a_generator.throw(MyException, 'messaggio di errore')</code>	<code>a_generator.throw(MyException('messaggio di errore'))</code>
③	<code>a_generator.throw('messaggio di errore')</code>	<i>non supportato</i>

1. Nella sua forma più semplice, un generatore lancia un'eccezione senza un messaggio di errore personalizzato. In questo caso la sintassi non è cambiata nel passaggio da Python 2 a Python 3.
2. Se il generatore lancia un'eccezione *con* un messaggio di errore personalizzato, allora dovete passare la stringa di errore all'eccezione quando la create.
3. Python 2 supportava anche il lancio di un'eccezione utilizzando *solo* un messaggio di errore personalizzato. Python 3 non lo supporta e lo script 2to3 visualizzerà un avvertimento dicendovi che avrete bisogno di correggere manualmente questo codice.

A.23. LA FUNZIONE GLOBALE xrange()

In Python 2 c'erano due modi per ottenere una lista di numeri: `range()`, che restituiva una lista, e `xrange()`, che restituiva un iteratore. In Python 3 `range()` restituisce un iteratore e `xrange()` non esiste più.

Note	Python 2	Python 3
①	<code>xrange(10)</code>	<code>range(10)</code>
②	<code>a_list = range(10)</code>	<code>a_list = list(range(10))</code>
③	<code>[i for i in xrange(10)]</code>	<code>[i for i in range(10)]</code>
④	<code>for i in range(10):</code>	<i>nessuna modifica</i>
⑤	<code>sum(range(10))</code>	<i>nessuna modifica</i>

1. Nel caso più semplice, lo script 2to3 convertirà semplicemente `xrange()` in `range()`.
2. Se il vostro programma in Python 2 usava `range()`, lo script 2to3 non sa se avete bisogno di una lista o se un iteratore potrebbe bastare. Pecca per eccesso di prudenza e converte il valore di ritorno in una lista chiamando la funzione `list()`.
3. Se la funzione `xrange()` si trova in una descrizione di lista, lo script 2to3 è abbastanza scaltro da *non* racchiudere la funzione `range()` in una invocazione a `list()`. La descrizione di lista lavora altrettanto bene con l'iteratore restituito dalla funzione `range()`.

4. Similmente, un ciclo `for` lavora altrettanto bene con un iteratore, quindi non c'è bisogno di cambiare nulla qui.
5. Anche la funzione `sum()` lavora altrettanto bene con un iteratore, quindi `2to3` non effettua alcun cambiamento neanche qui. Come per i metodi dei dizionari che restituiscono viste invece di liste, questo si applica a `min()`, `max()`, `sum()`, `list()`, `tuple()`, `set()`, `sorted()`, `any()` e `all()`.

A.24. LE FUNZIONI GLOBALI `raw_input()` E `input()`

Python 2 aveva due funzioni globali per chiedere all'utente di inserire dati in ingresso dalla riga di comando. La prima, chiamata `input()`, si aspettava che l'utente inserisse un'espressione Python (e ne restituiva il risultato). La seconda, chiamata `raw_input()`, restituiva semplicemente qualsiasi cosa l'utente avesse digitato. Questo comportamento era fonte di terribile confusione per i principianti e generalmente considerato come un "neo" del linguaggio. Python 3 asporta questo neo rinominando `raw_input()` a `input()`, in modo che funzioni come ci si aspetta intuitivamente che debba funzionare.

Note	Python 2	Python 3
①	<code>raw_input()</code>	<code>input()</code>
②	<code>raw_input('prompt')</code>	<code>input('prompt')</code>
③	<code>input()</code>	<code>eval(input())</code>

1. Nella sua forma più semplice, `raw_input()` diventa `input()`.
2. In Python 2 la funzione `raw_input()` poteva prendere come parametro un prompt, cioè una stringa di caratteri visualizzata per indicare l'attesa di un nuovo ingresso da parte della funzione. Questo parametro è stato mantenuto in Python 3.
3. Se avete effettivamente bisogno di chiedere all'utente un'espressione Python da valutare, usate la funzione `input()` e passatene il risultato a `eval()`.

A.25. GLI ATTRIBUTI `func_*` DELLE FUNZIONI

In Python 2 il codice all'interno delle funzioni può accedere a particolari attributi della funzione stessa. In Python 3 questi attributi speciali di una funzione sono stati rinominati per renderli consistenti con altri attributi.

Note	Python 2	Python 3
------	----------	----------

①	<code>a_function.func_name</code>	<code>a_function.__name__</code>
②	<code>a_function.func_doc</code>	<code>a_function.__doc__</code>
③	<code>a_function.func_defaults</code>	<code>a_function.__defaults__</code>
④	<code>a_function.func_dict</code>	<code>a_function.__dict__</code>
⑤	<code>a_function.func_closure</code>	<code>a_function.__closure__</code>
⑥	<code>a_function.func_globals</code>	<code>a_function.__globals__</code>
⑦	<code>a_function.func_code</code>	<code>a_function.__code__</code>

1. L'attributo `__name__` (precedentemente `func_name`) contiene il nome della funzione.
2. L'attributo `__doc__` (precedentemente `func_doc`) contiene la docstring che avete definito nel codice sorgente della funzione.
3. L'attributo `__defaults__` (precedentemente `func_defaults`) è una tupla contenente i valori predefiniti degli argomenti per quegli argomenti che hanno un valore predefinito.
4. L'attributo `__dict__` (precedentemente `func_dict`) è lo spazio di nomi che supporta la gestione di attributi arbitrari per la funzione.
5. L'attributo `__closure__` (precedentemente `func_closure`) è una tupla di celle che contengono i legami per le variabili libere della funzione.
6. L'attributo `__globals__` (precedentemente `func_globals`) è un riferimento allo spazio di nomi globale del modulo in cui la funzione è stata definita.
7. L'attributo `__code__` (precedentemente `func_code`) è un oggetto codice che rappresenta il corpo della funzione compilata.

A.26. IL METODO DI I/O `xreadlines()`

In Python 2 gli oggetti file avevano un metodo `xreadlines()` che restituiva un iteratore per leggere il file una riga alla volta. Questo era utile nei cicli `for` e in alcuni altri casi. In effetti, era così utile che le ultime versioni di Python 2 hanno aggiunto questa capacità agli oggetti file stessi.

In Python 3 il metodo `xreadlines()` non esiste più. `2to3` può correggere i casi più semplici, ma alcuni casi limite richiederanno un intervento manuale.

Note	Python 2	Python 3
①	<code>for line in a_file.xreadlines():</code>	<code>for line in a_file:</code>
②	<code>for line in a_file.xreadlines(5):</code>	<i>nessun cambiamento (guasto)</i>

1. Se di solito chiamavate `xreadlines()` senza argomenti, 2to3 convertirà la chiamata al solo oggetto file. In Python 3, questo è il modo in cui si effettua la stessa operazione: leggere il file una riga alla volta ed eseguire il corpo del ciclo `for`.
2. Se di solito chiamavate `xreadlines()` con un argomento (il numero di righe da leggere alla volta), 2to3 non vi correggerà e il vostro codice fallirà con un messaggio `AttributeError: '_io.TextIOWrapper' object has no attribute 'xreadlines'`. Potete manualmente sostituire `xreadlines()` con `readlines()` per farlo funzionare in Python 3. (Il metodo `readlines()` ora restituisce un iteratore, quindi è esattamente tanto efficiente quanto lo era `xreadlines()` in Python 2.)



A.27. FUNZIONI `lambda` CHE ACCETTANO UNA TUPLA INVECE DI PARAMETRI MULTIPLI

In Python 2 potevate definire una funzione `lambda` anonima che accettava diversi parametri scrivendola in modo che accettasse una tupla con uno specifico numero di elementi. In effetti, Python 2 avrebbe “spacchettato” la tupla negli argomenti con nome a cui poi avreste potuto riferirvi (attraverso il nome) all’interno della funzione `lambda`. In Python 3 potete ancora passare una tupla a una funzione `lambda`, ma l’interprete Python non spacchetterà la tupla in argomenti con nome. Invece, dovrete riferirvi a ogni argomento tramite l’indice della sua posizione.

Note	Python 2	Python 3
①	<code>lambda (x,): x + f(x)</code>	<code>lambda x1: x1[0] + f(x1[0])</code>
②	<code>lambda (x, y): x + f(y)</code>	<code>lambda x_y: x_y[0] + f(x_y[1])</code>
③	<code>lambda (x, (y, z)): x + y + z</code>	<code>lambda x_y_z: x_y_z[0] + x_y_z[1][0] + x_y_z[1][1]</code>
④	<code>lambda x, y, z: x + y + z</code>	<i>nessuna modifica</i>

1. Se avete definito una funzione `lambda` che accetta una tupla composta da un elemento, in Python 3 questa diventerà una `lambda` con riferimenti a `x1[0]`. Il nome `x1` viene generato automaticamente dallo script 2to3 sulla base degli argomenti con nome nella tupla originale.
2. La tupla di due elementi `(x, y)` argomento di una funzione `lambda` viene convertita a `x_y` con argomenti posizionali `x_y[0]` e `x_y[1]`.

3. Lo script 2to3 può perfino gestire funzioni lambda con tuple innestate di argomenti con nome. Il codice Python 3 risultante è leggermente illeggibile, ma funziona allo stesso modo in cui funzionava in Python 2.
4. Potete definire funzioni lambda che accettano più argomenti. Senza parentesi attorno agli argomenti, Python 2 le tratta semplicemente come funzioni lambda con diversi argomenti, in modo che all'interno della funzione lambda possiate riferirvi agli argomenti tramite il loro nome esattamente come avviene per ogni altra funzione. Questa sintassi funziona ancora in Python 3.

A.28. ATTRIBUTI SPECIALI DEI METODI

In Python 2 i metodi di classe possono fare riferimento all'oggetto classe in cui sono definiti così come all'oggetto metodo stesso. `im_self` è l'istanza dell'oggetto classe; `im_func` rappresenta l'oggetto funzione; `im_class` è la classe di `im_self`. In Python 3 questi attributi speciali dei metodi sono stati rinominati per seguire le convenzioni sui nomi degli altri attributi.

Note	Python 2	Python 3
	<code>aClassInstance.aClassMethod.im_func</code>	<code>aClassInstance.aClassMethod.__func__</code>
	<code>aClassInstance.aClassMethod.im_self</code>	<code>aClassInstance.aClassMethod.__self__</code>
	<code>aClassInstance.aClassMethod.im_class</code>	<code>aClassInstance.aClassMethod.__self__.__class__</code>

A.29. IL METODO SPECIALE `__nonzero__`

In Python 2 potevate costruire una vostra classe permettendole di essere usata in un contesto logico. Per esempio, potevate istanziare la classe e poi usare l'istanza in un'istruzione `if`. Per fare questo, definivate un metodo speciale `__nonzero__()` che restituiva `True` o `False` e veniva chiamato ogni volta che l'istanza era usata in un contesto logico. In Python 3 potete ancora farlo, ma il nome del metodo è stato cambiato in `__bool__()`.

Note	Python 2	Python 3
①	<pre>class A: def __nonzero__(self): pass</pre>	<pre>class A: def __bool__(self): pass</pre>
②	<pre>class A: def __nonzero__(self, x, y): pass</pre>	<i>nessuna modifica</i>

1. Al posto di `__nonzero__()`, Python 3 chiama il metodo `__bool__()` per valutare un'istanza in un contesto logico.
2. Comunque, se avete un metodo `__nonzero__()` che accetta argomenti, lo strumento 2to3 presumerà che lo stiate usando per qualche altro scopo e non effettuerà alcun cambiamento.

A.30. LETTERALI IN BASE OTTO

La sintassi per definire numeri in base 8 (ottali) è leggermente cambiata nel passaggio da Python 2 a Python 3.

Note	Python 2	Python 3
	<code>x = 0755</code>	<code>x = 0o755</code>

A.31. `sys.maxint`

A causa dell'integrazione tra i tipi `long` e `int`, la costante `sys.maxint` non è più accurata. Dato che il valore potrebbe ancora essere utile per determinare le capacità di una specifica piattaforma, la costante è stata mantenuta ma rinominata in `sys.maxsize`.

Note	Python 2	Python 3
①	<code>from sys import maxint</code>	<code>from sys import maxsize</code>
②	<code>a_function(sys.maxint)</code>	<code>a_function(sys.maxsize)</code>

1. `maxint` diventa `maxsize`.
2. Ogni uso di `sys.maxint` diventa `sys.maxsize`.

A.32. LA FUNZIONE GLOBALE `callable()`

In Python 2 potevate controllare se un oggetto era invocabile (come una funzione) tramite la funzione globale `callable()`. In Python 3 questa funzione globale è stata eliminata. Per controllare se un oggetto è invocabile, verificate l'esistenza del metodo speciale `__call__()`.

Note	Python 2	Python 3
	<code>callable(anything)</code>	<code>hasattr(anything, '__call__')</code>

A.33. LA FUNZIONE GLOBALE `zip()`

In Python 2 la funzione globale `zip()` prendeva un qualsiasi numero di sequenze e restituiva una lista di tuple. La prima tupla conteneva il primo elemento di ogni sequenza, la seconda tupla conteneva il secondo elemento di ogni sequenza, e così via. In Python 3 `zip()` restituisce un iteratore invece di una lista.

Note	Python 2	Python 3
①	<code>zip(a, b, c)</code>	<code>list(zip(a, b, c))</code>
②	<code>d.join(zip(a, b, c))</code>	<i>nessuna modifica</i>

1. Nella forma più semplice, potete ottenere il vecchio comportamento della funzione `zip()` circondandone il valore di ritorno con una chiamata a `list()`, che attraversa l'iteratore restituito da `zip()` e restituisce una lista reale dei risultati.
2. In contesti che iterano già attraverso tutti gli elementi di una sequenza (come questa chiamata al metodo `join()`) l'iteratore che `zip()` restituisce funzionerà adeguatamente. Lo script `2to3` è abbastanza scaltro da riconoscere questi casi e non fare alcun cambiamento al vostro codice.

A.34. L'ECCEZIONE DI TIPO `StandardError`

In Python 2 `StandardError` era la classe base per tutte le eccezioni built-in tranne `StopIteration`, `GeneratorExit`, `KeyboardInterrupt` e `SystemExit`. In Python 3 la classe `StandardError` è stata eliminata; usate `Exception` al suo posto.

Note	Python 2	Python 3
	<code>x = StandardError()</code>	<code>x = Exception()</code>
	<code>x = StandardError(a, b, c)</code>	<code>x = Exception(a, b, c)</code>

A.35. LE COSTANTI DEL MODULO `types`

Il modulo `types` contiene una varietà di costanti per aiutarvi a determinare il tipo di un oggetto. In Python 2 conteneva costanti per tutti i tipi primitivi come `dict` e `int`. In Python 3 queste costanti sono state eliminate; vi basta usare il nome del tipo primitivo.

Note	Python 2	Python 3
------	----------	----------

	<code>types.UnicodeType</code>	<code>str</code>
	<code>types.StringType</code>	<code>bytes</code>
	<code>types.DictType</code>	<code>dict</code>
	<code>types.IntType</code>	<code>int</code>
	<code>types.LongType</code>	<code>int</code>
	<code>types.ListType</code>	<code>list</code>
	<code>types.NoneType</code>	<code>type(None)</code>
	<code>types.BooleanType</code>	<code>bool</code>
	<code>types.BufferType</code>	<code>memoryview</code>
	<code>types.ClassType</code>	<code>type</code>
	<code>types.ComplexType</code>	<code>complex</code>
	<code>types.EllipsisType</code>	<code>type(Ellipsis)</code>
	<code>types.FloatType</code>	<code>float</code>
	<code>types.ObjectType</code>	<code>object</code>
	<code>types.NotImplementedType</code>	<code>type(NotImplemented)</code>
	<code>types.SliceType</code>	<code>slice</code>
	<code>types.TupleType</code>	<code>tuple</code>
	<code>types.TypeType</code>	<code>type</code>
	<code>types.XRangeType</code>	<code>range</code>

☞ `types.StringType` corrisponde a `bytes` invece che a `str` perché una “stringa” (non una stringa Unicode, semplicemente una stringa normale) in Python 2 è in realtà solo una sequenza di byte in una particolare codifica di carattere.

A.36. LA FUNZIONE GLOBALE `isinstance()`

La funzione `isinstance()` controlla se un oggetto è un'istanza di una particolare classe o tipo. In Python 2 potevate passare una tupla di tipi e `isinstance()` avrebbe restituito `True` se l'oggetto fosse stato di uno di quei tipi. In Python 3 potete ancora farlo, ma passare lo stesso tipo due volte è deprecato.

Note	Python 2	Python 3
------	----------	----------

	<code>isinstance(x, (int, float, int))</code>	<code>isinstance(x, (int, float))</code>
--	---	--

A.37. IL TIPO DI DATO `basestring`

Python 2 aveva due tipi di stringhe: Unicode e non Unicode. Ma c'era anche un altro tipo, `basestring`. Era un tipo astratto, una superclasse per entrambi i tipi `str` e `unicode`. Non poteva essere chiamata o istanziata direttamente, ma potevate passarla alla funzione globale `isinstance()` per controllare che un oggetto fosse una stringa Unicode o una stringa non Unicode. In Python 3 c'è un solo tipo di stringhe, quindi `basestring` non ha ragione di esistere.

Note	Python 2	Python 3
	<code>isinstance(x, basestring)</code>	<code>isinstance(x, str)</code>

A.38. IL MODULO `itertools`

Python 2.3 ha introdotto il modulo `itertools`, che definiva delle varianti per le funzioni globali `zip()`, `map()` e `filter()` che restituivano iteratori invece di liste. In Python 3 quelle funzioni globali restituiscono iteratori, quindi le corrispondenti funzioni del modulo `itertools` sono state eliminate. (Ci sono ancora molte funzioni utili nel modulo `itertools`, solo non queste.)

Note	Python 2	Python 3
①	<code>itertools.izip(a, b)</code>	<code>zip(a, b)</code>
②	<code>itertools.imap(a, b)</code>	<code>map(a, b)</code>
③	<code>itertools.ifilter(a, b)</code>	<code>filter(a, b)</code>
④	<code>from itertools import imap, izip, foo</code>	<code>from itertools import foo</code>

1. Invece di `itertools.izip()` usate semplicemente la funzione globale `zip()`.
2. Invece di `itertools.imap()` usate semplicemente `map()`.
3. `itertools.ifilter()` diventa `filter()`.
4. Il modulo `itertools` esiste ancora in Python 3, gli mancano solo le funzioni che sono state spostate nello spazio di nomi globale. Lo script `2to3` è abbastanza scaltro da rimuovere le specifiche istruzioni di `import` per moduli che non esistono più, lasciando intatte le altre.

A.39. `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback`

Python 2 aveva tre variabili nel modulo `sys` alle quali potevate accedere durante la gestione di un'eccezione: `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback`. (In realtà, esse risalgono a Python 1.) Fin da Python 1.5 queste variabili sono state deprecate a favore di `sys.exc_info()`, una funzione che restituisce una tupla contenente tutti e tre i valori. In Python 3 queste singole variabili sono infine sparite, perciò dovete usare la funzione `sys.exc_info()`.

Note	Python 2	Python 3
	<code>sys.exc_type</code>	<code>sys.exc_info()[0]</code>
	<code>sys.exc_value</code>	<code>sys.exc_info()[1]</code>
	<code>sys.exc_traceback</code>	<code>sys.exc_info()[2]</code>

A.40. DESCRIZIONI DI LISTA SULLE TUPLE

In Python 2, se volevate implementare una descrizione di lista che iterasse su una tupla, non avevate bisogno di mettere le parentesi attorno ai valori della tupla. In Python 3 le parentesi esplicite sono obbligatorie.

Note	Python 2	Python 3
	<code>[i for i in 1, 2]</code>	<code>[i for i in (1, 2)]</code>

A.41. LA FUNZIONE `os.getcwd()`

Python 2 aveva una funzione chiamata `os.getcwd()` che restituiva la directory di lavoro sotto forma di una stringa (non Unicode). Dato che i moderni file system possono gestire i nomi di directory in qualsiasi codifica di carattere, Python 2.3 ha introdotto `os.getcwdu()`. La funzione `os.getcwdu()` restituisce la directory di lavoro corrente come una stringa Unicode. In Python 3 c'è un solo tipo di stringhe (Unicode), quindi `os.getcwd()` è tutto quello di cui avete bisogno.

Note	Python 2	Python 3
	<code>os.getcwdu()</code>	<code>os.getcwd()</code>

A.42. METACLASSI

In Python 2 potevate creare metaclassi definendo l'argomento `metaclass` nella dichiarazione della classe, oppure definendo uno speciale attributo `__metaclass__` a livello di classe. In Python 3 questo attributo a livello di classe è stato eliminato.

Note	Python 2	Python 3
①	<pre>class C(metaclass=PapayaMeta): pass</pre>	<i>nessuna modifica</i>
②	<pre>class Whip: __metaclass__ = PapayaMeta</pre>	<pre>class Whip(metaclass=PapayaMeta): pass</pre>
③	<pre>class C(Whipper, Beater): __metaclass__ = PapayaMeta</pre>	<pre>class C(Whipper, Beater, metaclass=PapayaMeta): pass</pre>


1. Dichiarare la metaclassa nella dichiarazione della classe funzionava in Python 2 e funziona ancora allo stesso modo in Python 3.
2. Dichiarare la metaclassa in un attributo di classe funzionava in Python 2 ma non funziona in Python 3.
3. Lo script `2to3` è abbastanza potente da costruire una dichiarazione di classe valida anche se la classe eredita da una o più classi base.

A.43. QUESTIONI DI STILE

Il resto delle “correzioni” elencate qui di seguito non sono realmente correzioni vere e proprie. Vale a dire che le cose che modificano sono considerate questioni di stile, non di sostanza. Funzionano tanto bene in Python 3 quanto funzionavano in Python 2, ma gli sviluppatori di Python hanno un legittimo interesse nel rendere il codice Python più uniforme possibile. A questo scopo, esiste una [guida ufficiale allo stile in Python](#) che delinea — dolorosamente in dettaglio — tutti quei tipi di particolari minuziosi e pedanti di cui quasi certamente non vi interessa sapere nulla. E dato che `2to3` fornisce un'infrastruttura così buona per la conversione di codice Python da una forma in un'altra, gli autori si sono presi la briga di aggiungere alcune caratteristiche opzionali per migliorare la leggibilità dei vostri programmi Python.

A.43.1. LETTERALI `set()` (ESPLICITA)


In Python 2 l'unico modo di definire un letterale insieme nel vostro codice era chiamare `set(a_sequence)`. In Python 3 questo funziona ancora, ma un modo più chiaro di farlo è usare la nuova notazione per i letterali insieme: le parentesi graffe. Questa notazione funziona per tutto tranne gli insiemi vuoti, perché anche i dizionari usano le parentesi graffe, quindi `{}` è un dizionario vuoto, non un insieme vuoto.

 Lo script `2to3` non correggerà i letterali `set()` per default. Se volete abilitare questa correzione, specificate `-f set_literal` sulla riga di comando quando eseguite `2to3`.

Note	Prima	Dopo
	<code>set([1, 2, 3])</code>	<code>{1, 2, 3}</code>
	<code>set((1, 2, 3))</code>	<code>{1, 2, 3}</code>
	<code>set([i for i in a_sequence])</code>	<code>{i for i in a_sequence}</code>

A.43.2. LA FUNZIONE GLOBALE `buffer()` (ESPLICITA)


Gli oggetti Python implementati in C possono esportare una “interfaccia di buffer” che permette ad altro codice Python di leggere e scrivere direttamente un blocco di memoria. (Questo è esattamente potente e spaventoso quanto sembra.) In Python 3 `buffer()` è stata rinominata `memoryview()`. (È un po' più complicato di così, ma potete quasi certamente ignorare le differenze.)

 Lo script `2to3` non correggerà la funzione `buffer()` per default. Se volete abilitare questa correzione, specificate `-f buffer` sulla riga di comando quando eseguite `2to3`.

Note	Prima	Dopo
	<code>x = buffer(y)</code>	<code>x = memoryview(y)</code>

A.43.3. GLI SPAZI BIANCHI ATTORNO ALLE VIRGOLE (ESPLICITA)


Nonostante Python sia draconiano riguardo agli spazi bianchi per l'indentazione del codice, il linguaggio è in realtà piuttosto liberale riguardo agli spazi bianchi in altre aree. Nell'ambito di liste, tuple, insiemi e dizionari, gli spazi bianchi possono apparire prima e dopo le virgole senza sortire alcun effetto negativo. Tuttavia, la guida allo stile in Python afferma che le virgole dovrebbero essere precedute da zero spazi e seguite da uno spazio. Sebbene questo sia un problema puramente estetico (il codice funziona comunque, sia in Python 2 che in Python 3) lo script `2to3` può opzionalmente correggerlo per voi.

 Lo script `2to3` non correggerà gli spazi bianchi attorno alle virgole per default. Se volete abilitare questa correzione, specificate `-f wscomma` sulla riga di comando quando eseguite `2to3`.

Note	Prima	Dopo
	<code>a ,b</code>	<code>a, b</code>
	<code>{a :b}</code>	<code>{a: b}</code>

A.43.4. IDIOMI COMUNI (ESPLICITA)

Esiste un certo numero di idiomi comuni accumulati dalla comunità Python. Alcuni, come il ciclo `while 1:`, risalgono a Python 1. (Python non ha avuto un vero tipo booleano fino alla versione 2.3, così gli sviluppatori usavano `1` e `0`.) I moderni programmatori Python dovrebbero allenare le loro menti all'uso delle moderne versioni di questi idiomi.

 Lo script `2to3` non correggerà gli idiomi comuni per default. Se volete abilitare questa correzione, specificate `-f idioms` sulla riga di comando quando eseguite `2to3`.

Note	Prima	Dopo
	<pre>while 1: do_stuff()</pre>	<pre>while True: do_stuff()</pre>

	<code>type(x) == T</code>	<code>isinstance(x, T)</code>
	<code>type(x) is T</code>	<code>isinstance(x, T)</code>
	<code>a_list = list(a_sequence)</code> <code>a_list.sort()</code> <code>do_stuff(a_list)</code>	<code>a_list = sorted(a_sequence)</code> <code>do_stuff(a_list)</code>

APPENDICE B. NOMI DEI METODI SPECIALI

“ La mia specialità è avere ragione quando altri hanno torto. ”

— George Bernard Shaw

B.1. IMMERSIONE!

Vi sono già stati presentati, nel corso di questo libro, alcuni metodi speciali — metodi “magici” che Python invoca quando usate una certa sintassi. Sfruttando i metodi speciali, le vostre classi possono assumere il comportamento di insiemi, dizionari, funzioni, iteratori, o addirittura numeri! Questa appendice serve sia come riferimento per i metodi speciali che abbiamo già visto, sia come breve introduzione ad alcuni dei metodi speciali più esoterici.

B.2. LE BASI

Se avete letto l'introduzione alle classi, avete già visto il metodo speciale più comune: il metodo `__init__()`. La maggior parte delle classi che scrivo finiscono per avere bisogno di un qualche tipo di inizializzazione. Ci sono anche alcuni altri metodi speciali di base che si rivelano particolarmente utili quando dovete effettuare il debug delle vostre classi.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
①	inizializzare un'istanza	<code>x = MyClass()</code>	<code>x.__init__()</code>
②	la rappresentazione “ufficiale” sotto forma di stringa	<code>repr(x)</code>	<code>x.__repr__()</code>
③	la rappresentazione “informale” sotto forma di stringa	<code>str(x)</code>	<code>x.__str__()</code>
④	la rappresentazione “informale” sotto forma di array di byte	<code>bytes(x)</code>	<code>x.__bytes__()</code>
⑤	la rappresentazione sotto forma di stringa formattata	<code>format(x, format_spec)</code>	<code>x.__format__(format_spec)</code>

1. Il metodo `__init__()` viene invocato *dopo* che l'istanza è stata creata. Se volete controllare l'effettivo processo di creazione, usate il metodo `__new__()`.
2. Per convenzione, la stringa restituita dal metodo `__repr__()` dovrebbe essere un'espressione Python valida.
3. Il metodo `__str__()` viene invocato anche quando chiamate `print(x)`.
4. Il metodo `__bytes__()` è *una novità di Python 3*, dato che il tipo `bytes` è stato introdotto in questa versione del linguaggio.
5. Per convenzione, `format_spec` dovrebbe conformarsi al mini-linguaggio per le specifiche di formato. Il modulo `decimal` nella libreria standard di Python implementa il proprio metodo `__format__()`.

B.3. CLASSI CHE SI COMPORTANO COME ITERATORI

Nel capitolo sugli iteratori avete visto come implementare un iteratore da zero utilizzando i metodi `__iter__()` e `__next__()`.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
①	iterare attraverso una sequenza	<code>iter(seq)</code>	<code>seq.__iter__()</code>
②	ottenere il valore successivo da un iteratore	<code>next(seq)</code>	<code>seq.__next__()</code>
③	creare un iteratore in ordine inverso	<code>reversed(seq)</code>	<code>seq.__reversed__()</code>

1. Il metodo `__iter__()` viene invocato ogni volta che create un nuovo iteratore. È un buon posto per inizializzare valori che saranno poi necessari durante l'iterazione.
2. Il metodo `__next__()` viene invocato ogni volta che recuperate il valore successivo da un iteratore.
3. Il metodo `__reversed__()` è poco comune. Prende una sequenza esistente e restituisce un iteratore che produce gli elementi della sequenza in ordine inverso, dall'ultimo al primo.

Come avete visto nel capitolo sugli iteratori, un ciclo `for` può agire su un iteratore. In questo ciclo:

```
for x in seq:
    print(x)
```


Python 3 invocherà `seq.__iter__()` per creare un iteratore, poi invocherà il metodo `__next__()` su quell'iteratore per ottenere ogni singolo valore di `x`. Quando il metodo `__next__()` solleva un'eccezione di tipo `StopIteration`, il ciclo termina normalmente.

B.4. ATTRIBUTI CALCOLATI

Note	Voi volete...	Quindi scrivete...	E Python invoca...
①	ottenere (incondizionatamente) un attributo calcolato	<code>x.my_property</code>	<code>x.__getattr__('my_property')</code>
②	ottenere (alternativamente) un attributo calcolato	<code>x.my_property</code>	<code>x.__getattr__('my_property')</code>
③	impostare il valore di un attributo	<code>x.my_property = value</code>	<code>x.__setattr__('my_property', value)</code>
④	eliminare un attributo	<code>del x.my_property</code>	<code>x.__delattr__('my_property')</code>
⑤	elencare tutti gli attributi e i metodi	<code>dir(x)</code>	<code>x.__dir__()</code>

1. Se la vostra classe definisce un metodo `__getattr__()`, Python lo invocherà *ogni volta che fate riferimento al nome di un attributo o di un metodo qualsiasi* (a parte i nomi dei metodi speciali, dato che questo causerebbe uno spiacevole ciclo infinito).
2. Se la vostra classe definisce un metodo `__getattr__()`, Python lo invocherà solo dopo aver cercato l'attributo in tutti i posti in cui viene cercato normalmente. Se un'istanza `x` definisce un attributo `color`, l'accesso a `x.color` *non* causerà l'invocazione di `x.__getattr__('color')` ma restituirà semplicemente il valore di `x.color` già definito.
3. Il metodo `__setattr__()` viene invocato ogni volta che assegnate un valore a un attributo.
4. Il metodo `__delattr__()` viene invocato ogni volta che eliminate un attributo.
5. Il metodo `__dir__()` è utile se definite un metodo `__getattr__()` o un metodo `__getattr__()`. Generalmente, l'invocazione di `dir(x)` elenca solo gli attributi e i metodi normali. Se il vostro metodo `__getattr__()` gestisce dinamicamente un attributo `color`, `dir(x)` non mostrerà `color` come uno degli attributi disponibili. Fornire una vostra implementazione del metodo `__dir__()` vi permette di mostrare `color` nell'elenco degli attributi disponibili, venendo in aiuto ad altri programmatori che vorrebbero usare la vostra classe senza doverne esaminare i dettagli interni.

La distinzione tra i metodi `__getattr__()` e `__getattribute__()` è sottile ma importante. Posso spiegarla con due esempi:

```
class Dynamo:
    def __getattr__(self, key):
        if key == 'color':          ①
            return 'PapayaWhip'
        else:
            raise AttributeError    ②

>>> dyn = Dynamo()

>>> dyn.color                      ③
'PapayaWhip'

>>> dyn.color = 'LemonChiffon'

>>> dyn.color                      ④
'LemonChiffon'
```


1. Il nome dell'attributo viene passato al metodo `__getattr__()` sotto forma di stringa. Se il nome è `'color'`, il metodo restituisce un valore. (In questo caso il valore è solo una stringa costante, ma normalmente effettuereste una qualche sorta di computazione e ne restituireste il risultato.)
2. Se il nome dell'attributo è sconosciuto, il metodo `__getattr__()` deve sollevare un'eccezione di tipo `AttributeError`, altrimenti il vostro codice fallirà silenziosamente quando accedete ad attributi non definiti. (Tecnicamente, se il metodo non solleva un'eccezione o restituisce esplicitamente un valore, allora restituisce `None`, il valore nullo di Python. In questo caso *tutti* gli attributi non esplicitamente definiti varrebbero `None`, che quasi sicuramente non è ciò che volete.)
3. L'istanza `dyn` non ha un attributo chiamato `color`, quindi il metodo `__getattr__()` viene invocato per fornirne il valore calcolato.
4. Dopo aver esplicitamente impostato `dyn.color`, il metodo `__getattr__()` non viene più invocato per fornire il valore di `dyn.color` perché `dyn.color` è già definito in questa istanza.

D'altra parte, il metodo `__getattribute__()` viene invocato in maniera assoluta e incondizionata.

```
class SuperDynamo:
    def __getattr__(self, key):
        if key == 'color':
            return 'PapayaWhip'
        else:
            raise AttributeError
```

```
>>> dyn = SuperDynamo()
>>> dyn.color                                ①
'PapayaWhip'
>>> dyn.color = 'LemonChiffon'
>>> dyn.color                                ②
'PapayaWhip'
```

1. Il metodo `__getattr__()` viene invocato per fornire il valore di `dyn.color`.
2. Anche dopo aver esplicitamente impostato `dyn.color`, il metodo `__getattr__()` viene ancora invocato per fornirne il valore. Se esiste, il metodo `__getattr__()` viene invocato incondizionatamente per ogni accesso ad attributi e metodi, anche per attributi che avete esplicitamente impostato dopo la creazione di un'istanza.

 Se la vostra classe definisce un metodo `__getattr__()`, vorrete probabilmente definire anche un metodo `__setattr__()` e coordinare i due per tenere traccia del valore degli attributi. Altrimenti, qualsiasi attributo che impostiate dopo aver creato un'istanza scomparirà in un buco nero.

Dovete usare il metodo `__getattr__()` con particolare attenzione, perché viene invocato anche quando Python cerca il nome di un metodo nella vostra classe.

```

class Rastan:
    def __getattr__(self, key):
        raise AttributeError ①
    def swim(self):
        pass

>>> hero = Rastan()

>>> hero.swim() ②

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __getattr__
AttributeError

```

1. Questa classe definisce un metodo `__getattr__()` che solleva sempre un'eccezione di tipo `AttributeError`. Nessun accesso ad attributi o metodi avrà successo.
2. Quando invocate `hero.swim()`, Python cerca un metodo `swim()` nella classe `Rastan`. Questa ricerca passa attraverso il metodo `__getattr__()`, *perché tutti gli accessi ad attributi e metodi passano attraverso il metodo `__getattr__()`*. In questo caso il metodo `__getattr__()` solleva un'eccezione di tipo `AttributeError`, quindi l'accesso al metodo fallisce, quindi la chiamata di metodo fallisce.

B.5. CLASSI CHE SI COMPORTANO COME FUNZIONI

Potete rendere invocabile un'istanza di una classe — esattamente come è invocabile una funzione — definendo il metodo `__call__()`.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	“invocare” un'istanza come una funzione	<code>my_instance()</code>	<code>my_instance.__call__()</code>

Il modulo `zipfile` usa questa tecnica per definire una classe che possa decodificare un file zip criptato con una certa password. L'algoritmo di decodifica per i file zip richiede di memorizzare uno stato durante l'esecuzione. Definire il decodificatore come una classe vi permette di mantenere questo stato all'interno di una singola istanza di tale classe. Lo stato viene inizializzato nel metodo `__init__()` e aggiornato man mano

che il file viene decodificato. Ma dato che la classe è anche “invocabile” come una funzione, potete passare l’istanza come primo argomento alla funzione `map()` in questo modo:

```
# estratto da zipfile.py
class _ZipDecrypter:
    .
    .
    .

    def __init__(self, pwd):
        self.key0 = 305419896          ①
        self.key1 = 591751049
        self.key2 = 878082192
        for p in pwd:
            self._UpdateKeys(p)

    def __call__(self, c):              ②
        assert isinstance(c, int)
        k = self.key2 | 2
        c = c ^ (((k * (k^1)) >> 8) & 255)
        self._UpdateKeys(c)
        return c

    .
    .
    .

zd = _ZipDecrypter(pwd)                ③
bytes = zef_file.read(12)
h = list(map(zd, bytes[0:12]))          ④
```

1. La classe `_ZipDecrypter` mantiene il proprio stato sotto forma di tre chiavi di cifratura, che vengono successivamente aggiornate facendone ruotare i bit nel metodo `_UpdateKeys()` (che qui non viene mostrato).
2. La classe definisce un metodo `__call__()` che rende le istanze di questa classe invocabili come funzioni. In questo caso il metodo `__call__()` decodifica un singolo byte del file zip, quindi aggiorna le chiavi di cifratura sulla base del byte che è stato decodificato.

3. `zd` è un'istanza della classe `_ZipDecrypter`. La variabile `pwd` viene passata al metodo `__init__()`, dove viene memorizzata e usata per aggiornare le chiavi di cifratura per la prima volta.
4. Dati i primi 12 byte di un file zip, li decodificate applicando `zd` ai byte, in sostanza “invocando” 12 volte `zd`, quindi invocando 12 volte il metodo `__call__()`, che per 12 volte aggiorna lo stato interno del decodificatore e restituisce un byte come risultato.

B.6. CLASSI CHE SI COMPORTANO COME INSIEMI

Nel caso la vostra classe agisca come contenitore per un insieme di valori — cioè nel caso abbia senso chiedere se la vostra classe “contiene” un valore — allora dovrebbe probabilmente definire i seguenti metodi speciali che le permettono di assumere il comportamento di un insieme.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	il numero di elementi	<code>len(s)</code>	<code>s.__len__()</code>
	sapere se contiene uno specifico valore	<code>x in s</code>	<code>s.__contains__(x)</code>

Il modulo `cgi` sfrutta questi metodi nella sua classe `FieldStorage`, che rappresenta tutti i campi di una form o i parametri di richiesta inviati a una pagina web dinamica.

```

# Uno script che risponde a http://example.com/search?q=cgi
import cgi
fs = cgi.FieldStorage()

if 'q' in fs: ①
    do_search()

# Un estratto da cgi.py che spiega come funziona lo script
class FieldStorage:
    .
    .
    .

    def __contains__(self, key): ②
        if self.list is None:
            raise TypeError('not indexable')

        return any(item.name == key for item in self.list) ③

    def __len__(self): ④
        return len(self.keys()) ⑤

```

1. Una volta creata un'istanza della classe `cgi.FieldStorage`, potete usare l'operatore “in” per controllare se un particolare parametro è stato incluso nella stringa di richiesta.
2. Il metodo `__contains__()` è la magia che fa funzionare quel controllo.
3. Quando dite `if 'q' in fs`, Python cerca nell'oggetto `fs` il metodo `__contains__()`, che è definito in `cgi.py`. Il valore `'q'` è passato al metodo `__contains__()` come argomento `key`.
4. La stessa classe `FieldStorage` è in grado di restituire il valore della propria lunghezza, quindi `len(fs)` invocherà il metodo `__len__()` sull'istanza della classe `FieldStorage` per restituire il numero di parametri di richiesta che sono stati identificati.
5. Il metodo `self.keys()` verifica se `self.list is None`, quindi il metodo `__len__()` non ha bisogno di duplicare questo controllo di errore.

B.7. CLASSI CHE SI COMPORTANO COME DIZIONARI

Estendendo un poco la sezione precedente, potete definire classi che non solo rispondono all'operatore "in" e alla funzione `len()`, ma che si comportano come dizionari veri e propri, restituendo valori sulla base di chiavi.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	ottenere un valore tramite la sua chiave	<code>x[key]</code>	<code>x.__getitem__(key)</code>
	impostare un valore tramite la sua chiave	<code>x[key] = value</code>	<code>x.__setitem__(key, value)</code>
	eliminare una coppia chiave-valore	<code>del x[key]</code>	<code>x.__delitem__(key)</code>
	fornire un valore predefinito per chiavi mancanti	<code>x[nonexistent_key]</code>	<code>x.__missing__(nonexistent_key)</code>

La classe `FieldStorage` del modulo `cgi` definisce anche questi metodi speciali in modo che possiate fare cose come queste:


```
# Uno script che risponde a http://example.com/search?q=cgi
import cgi
fs = cgi.FieldStorage()
if 'q' in fs:
    do_search(fs['q'])
```

①

```
# Un estratto da cgi.py che spiega come funziona lo script
class FieldStorage:
```

```
.
.
.
```

```
def __getitem__(self, key):
    if self.list is None:
        raise TypeError('not indexable')
    found = []
    for item in self.list:
        if item.name == key: found.append(item)
    if not found:
        raise KeyError(key)
    if len(found) == 1:
        return found[0]
    else:
        return found
```

②

1. L'oggetto `fs` è un'istanza di `cgi.FieldStorage`, tuttavia potete valutare espressioni come `fs['q']`.
2. `fs['q']` invoca il metodo `__getitem__()` con il parametro `key` impostato a `'q'`. Quindi controlla se la lista di parametri di richiesta che mantiene internamente (`self.list`) contiene un elemento il cui attributo `name` corrisponde alla chiave `data`.

B.8. CLASSI CHE SI COMPORTANO COME NUMERI

Usando i metodi speciali appropriati, potete definire classi che si comportano come numeri. Questo significa che potete sommarle, sottrarle ed eseguire su di esse altre operazioni matematiche. Questo è il modo in cui

sono realizzate le frazioni — la classe `Fraction` implementa questi metodi speciali in modo che possiate fare cose come questa:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> x / 3
Fraction(1, 9)
```

Ecco la lista completa dei metodi speciali che dovete implementare per avere una classe simile ai numeri.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	addizione	$x + y$	<code>x.__add__(y)</code>
	sottrazione	$x - y$	<code>x.__sub__(y)</code>
	moltiplicazione	$x * y$	<code>x.__mul__(y)</code>
	divisione	x / y	<code>x.__truediv__(y)</code>
	divisione intera	$x // y$	<code>x.__floordiv__(y)</code>
	modulo (resto)	$x \% y$	<code>x.__mod__(y)</code>
	divisione intera & modulo	<code>divmod(x, y)</code>	<code>x.__divmod__(y)</code>
	elevamento a potenza	$x ** y$	<code>x.__pow__(y)</code>
	scorrimento a sinistra	$x \ll y$	<code>x.__lshift__(y)</code>
	scorrimento a destra	$x \gg y$	<code>x.__rshift__(y)</code>
	and bit per bit	$x \& y$	<code>x.__and__(y)</code>
	xor bit per bit	$x \wedge y$	<code>x.__xor__(y)</code>
	or bit per bit	$x y$	<code>x.__or__(y)</code>

Tutto questo va bene se `x` è un'istanza di una classe che implementa questi metodi. Ma cosa succede se non ne implementa uno? O peggio, cosa succede se lo implementa, ma non può gestire certi tipi di argomenti?

Per esempio:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> 1 / x
Fraction(3, 1)
```

Questo *non* è il caso in cui un oggetto `Fraction` viene diviso per un intero (come nell'esempio precedente). Quel caso era semplice: `x / 3` invoca `x.__truediv__(3)` e il metodo `__truediv__()` della classe `Fraction` gestisce tutta la matematica coinvolta. Ma gli interi non “sanno” come fare operazioni aritmetiche con le frazioni. Quindi come mai questo esempio funziona?

Esiste un secondo insieme di metodi speciali per l'aritmetica con *operandi rovesciati*. Data un'operazione aritmetica con due operandi (e.g. `x / y`), ci sono due modi di eseguirla:

1. dire a `x` di dividersi per `y`, oppure
2. dire a `y` di dividere `x` per lei stessa.

L'insieme di metodi speciali appena visto adotta il primo approccio: dato `x / y`, fornisce a `x` un modo per dire: “So come dividermi per `y`.” L'insieme di metodi speciali che segue adotta il secondo approccio, fornendo a `y` un modo per dire: “So come fare il denominatore e dividere `x` per me stessa.”

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	addizione	<code>x + y</code>	<code>y.__radd__(x)</code>
	sottrazione	<code>x - y</code>	<code>y.__rsub__(x)</code>
	moltiplicazione	<code>x * y</code>	<code>y.__rmul__(x)</code>
	divisione	<code>x / y</code>	<code>y.__rtruediv__(x)</code>
	divisione intera	<code>x // y</code>	<code>y.__rfloordiv__(x)</code>
	modulo (resto)	<code>x % y</code>	<code>y.__rmod__(x)</code>
	divisione intera & modulo	<code>divmod(x, y)</code>	<code>y.__rdivmod__(x)</code>
	elevamento a potenza	<code>x ** y</code>	<code>y.__rpow__(x)</code>
	scorrimento a sinistra	<code>x << y</code>	<code>y.__rlshift__(x)</code>
	scorrimento a destra	<code>x >> y</code>	<code>y.__rrshift__(x)</code>
	and bit per bit	<code>x & y</code>	<code>y.__rand__(x)</code>
	xor bit per bit	<code>x ^ y</code>	<code>y.__rxor__(x)</code>
	or bit per bit	<code>x y</code>	<code>y.__ror__(x)</code>

Ma aspettate! C'è di più! Se state eseguendo operazioni “in loco”, come `x /= 3`, ci sono ancora altri metodi speciali aggiuntivi che potete definire.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	addizione in loco	<code>x += y</code>	<code>x.__iadd__(y)</code>
	sottrazione in loco	<code>x -= y</code>	<code>x.__isub__(y)</code>
	moltiplicazione in loco	<code>x *= y</code>	<code>x.__imul__(y)</code>
	divisione in loco	<code>x /= y</code>	<code>x.__itruediv__(y)</code>
	divisione intera in loco	<code>x //= y</code>	<code>x.__ifloordiv__(y)</code>
	modulo in loco	<code>x %= y</code>	<code>x.__imod__(y)</code>
	elevamento a potenza in loco	<code>x **= y</code>	<code>x.__ipow__(y)</code>
	scorrimento a sinistra in loco	<code>x <<= y</code>	<code>x.__ilshift__(y)</code>
	scorrimento a destra in loco	<code>x >>= y</code>	<code>x.__irshift__(y)</code>
	and bit per bit in loco	<code>x &= y</code>	<code>x.__iand__(y)</code>
	xor bit per bit in loco	<code>x ^= y</code>	<code>x.__ixor__(y)</code>
	or bit per bit in loco	<code>x = y</code>	<code>x.__ior__(y)</code>

Notate che la maggior parte dei metodi per le operazioni in loco non è richiesta. Se non definite il metodo per una particolare operazione in loco, Python proverà a utilizzare gli altri metodi. Per esempio, cercando di eseguire l'espressione `x /= y`, Python proverà a...

1. Invocare `x.__itruediv__(y)`. Se questo metodo è definito e restituisce un valore diverso da `NotImplemented`, abbiamo finito.
2. Invocare `x.__truediv__(y)`. Se questo metodo è definito e restituisce un valore diverso da `NotImplemented`, il vecchio valore di `x` viene scartato e rimpiazzato dal valore restituito, esattamente come se aveste eseguito `x = x / y`.
3. Invocare `y.__rtruediv__(x)`. Se questo metodo è definito e restituisce un valore diverso da `NotImplemented`, il vecchio valore di `x` viene scartato e rimpiazzato dal valore restituito.

Quindi avete bisogno di definire metodi in loco come il metodo `__itruediv__()` solo se volete effettuare qualche ottimizzazione speciale per l'operando in loco. Altrimenti, Python essenzialmente riformulerà l'operazione in loco in modo da utilizzare un operando normale più un assegnamento di variabile.

Ci sono anche alcune operazioni matematiche “unarie” che gli oggetti simili a numeri possono eseguire su sé stessi.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	un numero negativo	-x	x.__neg__()
	un numero positivo	+x	x.__pos__()
	il valore assoluto	abs(x)	x.__abs__()
	l'inverso	~x	x.__invert__()
	un numero complesso	complex(x)	x.__complex__()
	un numero intero	int(x)	x.__int__()
	un numero in virgola mobile	float(x)	x.__float__()
	un numero arrotondato all'intero più vicino	round(x)	x.__round__()
	un numero arrotondato alla n-sima cifra	round(x, n)	x.__round__(n)
	il più piccolo intero che sia $\geq x$	math.ceil(x)	x.__ceil__()
	il più grande intero che sia $\leq x$	math.floor(x)	x.__floor__()
	troncare x all'intero più vicino verso lo 0	math.trunc(x)	x.__trunc__()
PEP 357	usare un numero come indice di lista	a_list[x]	a_list[x.__index__()]

B.9. CLASSI CHE POSSONO ESSERE CONFRONTATE

Ho separato questa sezione dalla precedente perché i confronti non sono strettamente un privilegio dei numeri. Molti tipi di dato possono essere confrontati — stringhe, liste, persino dizionari. Se state creando una vostra classe e ha senso confrontare i vostri oggetti con altri oggetti, potete usare i metodi speciali che seguono per realizzare i confronti.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	uguaglianza	x == y	x.__eq__(y)
	disuguaglianza	x != y	x.__ne__(y)
	minore di	x < y	x.__lt__(y)
	minore o uguale a	x <= y	x.__le__(y)
	maggiore di	x > y	x.__gt__(y)

	maggiore o uguale a	<code>x >= y</code>	<code>x.__ge__(y)</code>
	un valore di verità in un contesto logico	<code>if x:</code>	<code>x.__bool__()</code>

☞ Se definite un metodo `__lt__()` ma non definite un metodo `__gt__()`, Python userà il metodo `__lt__()` con gli operandi invertiti. Tuttavia, Python non combinerà i metodi. Per esempio, se definite un metodo `__lt__()` e un metodo `__eq__()` e provate a verificare se `x <= y`, Python non invocherà `__lt__()` e `__eq__()` in sequenza, ma invocherà soltanto il metodo `__le__()`.

B.10. CLASSI CHE POSSONO ESSERE SERIALIZZATE

Python supporta la serializzazione e la deserializzazione di oggetti arbitrari. (La maggior parte del materiale di consultazione su Python chiama questi processi “pickling” e “unpickling”.) La serializzazione può rivelarsi utile per salvare lo stato di un oggetto su un file e ripristinarlo successivamente. Tutti i tipi di dato nativi includono già il supporto per la serializzazione. Se create una classe che volete essere in grado di serializzare, vi suggerisco di leggere la documentazione sul protocollo di serializzazione per vedere quando e come i metodi speciali che seguono vengono invocati.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	la copia di un oggetto	<code>copy.copy(x)</code>	<code>x.__copy__()</code>
	la copia in profondità di un oggetto	<code>copy.deepcopy(x)</code>	<code>x.__deepcopy__()</code>
	ottenere lo stato di un oggetto prima della serializzazione	<code>pickle.dump(x, file)</code>	<code>x.__getstate__()</code>
	serializzare un oggetto	<code>pickle.dump(x, file)</code>	<code>x.__reduce__()</code>
	serializzare un oggetto (con un nuovo protocollo di pickling)	<code>pickle.dump(x, file, protocol_version)</code>	<code>x.__reduce_ex__(protocol_version)</code>

*	controllare come un oggetto viene creato durante la deserializzazione	<code>x = pickle.load(file)</code>	<code>x.__getnewargs__()</code>
*	ripristinare lo stato di un oggetto dopo la deserializzazione	<code>x = pickle.load(file)</code>	<code>x.__setstate__()</code>

* Per ricreare un oggetto serializzato, Python deve prima creare un nuovo oggetto simile all'oggetto serializzato e poi impostare i valori di tutti gli attributi del nuovo oggetto. Il metodo `__getnewargs__()` controlla come l'oggetto viene creato, poi il metodo `__setstate__()` controlla come i valori degli attributi vengono ripristinati.

B.11. CLASSI CHE POSSONO ESSERE USATE IN UN BLOCCO `with`

Un blocco `with` definisce un contesto di esecuzione; voi “entrate” nel contesto quando l'istruzione `with` viene eseguita e “uscite” dal contesto dopo che l'ultima istruzione nel blocco è stata eseguita.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	fare qualcosa di speciale quando entrate in un blocco <code>with</code>	<code>with x:</code>	<code>x.__enter__()</code>
	fare qualcosa di speciale quando uscite da un blocco <code>with</code>	<code>with x:</code>	<code>x.__exit__(exc_type, exc_value, traceback)</code>

Ecco come funziona l'idioma `with file`.

```
# estratto da io.py:
def _checkClosed(self, msg=None):
    '''Internal: raise an ValueError if file is closed
    ...

    if self.closed:
        raise ValueError('I/O operation on closed file.'
                           if msg is None else msg)

def __enter__(self):
    '''Context management protocol. Returns self.'''
    self._checkClosed() ①
    return self ②

def __exit__(self, *args):
    '''Context management protocol. Calls close()'''
    self.close() ③
```

1. L'oggetto file definisce sia un metodo `__enter__()` che un metodo `__exit__()`. Il metodo `__enter__()` verifica che il file sia aperto: se non lo è, il metodo `_checkClosed()` solleva un'eccezione.
2. Il metodo `__enter__()` dovrebbe quasi sempre restituire `self` — questo è l'oggetto che il blocco `with` userà per inoltrare gli accessi ad attributi e metodi.
3. All'uscita dal blocco `with`, l'oggetto file viene automaticamente chiuso. In che modo? Invocando `self.close()` nel metodo `__exit__()`.



Il metodo `__exit__()` verrà sempre invocato, anche se un'eccezione viene sollevata all'interno del blocco `with`. In effetti, nel caso in cui un'eccezione venga sollevata, le informazioni su quella eccezione saranno passate al metodo `__exit__()`. Si veda la documentazione sui [gestori di contesto per l'istruzione `with`](#) per maggiori dettagli.

Per ulteriori informazioni sui gestori di contesto, si veda [Chiudere i file automaticamente](#) e [Redirigere il canale standard di uscita](#).

B.12. ROBA VERAMENTE ESOTERICA

Se sapete quello che state facendo, potete ottenere il controllo quasi completo su come le classi sono confrontate, su come gli attributi sono definiti e su quali tipi di classe sono considerati sottotipi della vostra classe.

Note	Voi volete...	Quindi scrivete...	E Python invoca...
	il costruttore di una classe	<code>x = MyClass()</code>	<code>x.__new__()</code>
*	il distruttore di una classe	<code>del x</code>	<code>x.__del__()</code>
	che solo uno specifico insieme di attributi sia definito		<code>x.__slots__()</code>
	un valore di hash personalizzato	<code>hash(x)</code>	<code>x.__hash__()</code>
	ottenere il valore di una proprietà	<code>x.color</code>	<code>type(x).__dict__['color'].__get__(x, type(x))</code>
	impostare il valore di una proprietà	<code>x.color = 'PapayaWhip'</code>	<code>type(x).__dict__['color'].__set__(x, 'PapayaWhip')</code>
	eliminare una proprietà	<code>del x.color</code>	<code>type(x).__dict__['color'].__del__(x)</code>
	controllare se un oggetto è istanza della vostra classe	<code>isinstance(x, MyClass)</code>	<code>MyClass.__instancecheck__(x)</code>
	controllare se una classe è sottoclasse di una vostra classe	<code>issubclass(C, MyClass)</code>	<code>MyClass.__subclasscheck__(C)</code>
	controllare se una classe è sottoclasse di una vostra classe astratta di base	<code>issubclass(C, MyABC)</code>	<code>MyABC.__subclasshook__(C)</code>

* Determinare esattamente quando Python invoca il metodo speciale `__del__()` è un'operazione incredibilmente complicata. Per capirla appieno, avete bisogno di sapere come Python tiene traccia degli oggetti in memoria. Ecco un buon articolo sulla garbage collection in Python e sulla sua relazione con i metodi distruttori di classe. Dovreste anche leggere qualcosa sui riferimenti deboli, sul modulo `weakref` e probabilmente, per sicurezza, anche sul modulo `gc`.

B.13. LETTURE DI APPROFONDIMENTO

Moduli menzionati in questa appendice:

- Il modulo zipfile
- Il modulo cgi
- Il modulo collections
- Il modulo math
- Il modulo pickle
- Il modulo copy
- Il modulo abc (“Classi di Base Astratte”)

Altre letture divertenti:

- Mini-linguaggio per le specifiche di formato
- Modello dei dati di Python
- Tipi built-in
- PEP 357: Fare in modo di poter affettare qualsiasi oggetto
- PEP 3119: Introduzione alle classi di base astratte

APPENDICE C. DOVE PROSEGUIRE DA QUI

“Vai avanti per la tua strada, poiché esiste solo grazie al tuo cammino.”
— Sant’Agostino d’Ippona (attribuita)

C.1. COSE DA LEGGERE

Zac, questo libro deve terminare da qualche parte, e sfortunatamente non mi è possibile trattare tutti gli aspetti di Python 3. Per fortuna, esistono molte guide valide liberamente disponibili in rete. (Tutte le risorse elencate sono in lingua inglese.)

Decoratori:

- [Decoratori di funzione di Ariel Ortiz](#)
- [Ulteriori informazioni sui decoratori di funzione di Ariel Ortiz](#)
- [Incantare Python: i decoratori facilitano la magia di David Mertz](#)
- [Le definizioni di funzione nella documentazione ufficiale di Python](#)

Proprietà:

- [La funzione predefinita `property` di Python di Adam Goma](#)
- [Metodi per recuperare/impostare/incasinare valori di Ryan Tomayko](#)
- [La funzione `property\(\)` nella documentazione ufficiale di Python](#)

Descrittori:

- [Guida pratica ai descrittori di Raymond Hettinger](#)
- [Incantare Python: l’eleganza e le imperfezioni di Python, parte 2 di David Mertz](#)
- [I descrittori Python di Mark Summerfield](#)
- [Invocare i descrittori nella documentazione ufficiale di Python](#)

Uso di thread & processi:

- [Il modulo threading](#)
- [threading — Gestire thread concorrenti](#)
- [Il modulo multiprocessing](#)
- [multiprocessing — Gestire i processi come i thread](#)
- [I thread Python e il Global Interpreter Lock \(letteralmente, Blocco Globale dell'Interprete\) di Jesse Noller](#)
- [Dentro al GIL di Python \(video\) di David Beazley](#)

Metaclassi:

- [La programmazione con le metaclassi in Python di David Mertz e Michele Simionato](#)
- [La programmazione con le metaclassi in Python, parte 2 di David Mertz e Michele Simionato](#)
- [La programmazione con le metaclassi in Python, parte 3 di David Mertz e Michele Simionato](#)

[Il modulo Python della settimana](#), infine, è un sito gestito da Doug Hellman che si rivela essere una guida fantastica a molti dei moduli contenuti nella libreria standard di Python.

C.2. DOVE CERCARE CODICE COMPATIBILE CON PYTHON 3

Dato che Python 3 è relativamente nuovo, c'è una certa scarsità di librerie compatibili. Ecco alcuni dei posti dove cercare codice che funziona con Python 3.

- [Indice dei Pacchetti Python: lista dei pacchetti per Python 3](#)
- [Ricettario di Python: lista delle ricette etichettate “python3”](#)
- [Google Project Hosting: lista dei progetti etichettati “python3”](#)
- [SourceForge: lista dei progetti che corrispondono a “Python 3”](#)
- [GitHub: lista dei progetti che corrispondono a “python3” \(e lista dei progetti che corrispondono a “python 3”\)](#)
- [Bitbucket: lista dei progetti che corrispondono a “python3” \(e quelli che corrispondono a “python 3”\)](#)

© 2001–9 [Mark Pilgrim](#)

© 2009 [Giulio Piancastelli](#) per la traduzione italiana